PUCRS

ESCOLA POLITÉCNICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

LEONARDO GIBROWSKI FAÉ

A HIGH-LEVEL DSL IN RUST FOR EXPRESSING LINEAR PIPELINES ON MULTI-CORES, CLUSTERS AND GPUS

Porto Alegre
2025

PÓS-GRADUAÇÃO - *STRICTO SENSU*

Pontifícia Universidade Católica
do Rio Grande do Sul

PONTIFICAL CATHOLIC UNIVERSITY OF RIO GRANDE DO SUL
SCHOOL OF TECHNOLOGY
COMPUTER SCIENCE GRADUATE PROGRAM

A HIGH-LEVEL DSL IN RUST FOR EXPRESSING LINEAR PIPELINES ON MULTI-CORES, CLUSTERS AND GPUS

LEONARDO GIBROWSKI FAÉ

Master Thesis submitted to the Pontifical Catholic University of Rio Grande do Sul in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Prof. Dr. Dalvan Jair Griebler

Porto Alegre
2025

Ficha Catalográfica

F147h Faé, Leonardo Gibrowski

A High-Level DSL in Rust for Expressing Linear Pipelines on Multi-cores, Clusters and GPUs / Leonardo Gibrowski Faé. – 2025.

105.

Dissertação (Mestrado) – Programa de Pós-Graduação em Ciência da Computação, PUCRS.

Orientador: Prof. Dr. Dalvan Jair Griebler.

1. Rust. 2. Stream Processing. 3. SPar. 4. Code Transformation. 5. Parallel. I. Griebler, Dalvan Jair. II. Título.

Elaborada pelo Sistema de Geração Automática de Ficha Catalográfica da PUCRS com os dados fornecidos pelo(a) autor(a).

Bibliotecária responsável: Clarissa Jesinska Selbach CRB-10/2051

LEONARDO GIBROWSKI FAÉ

A HIGH-LEVEL DSL IN RUST FOR EXPRESSING LINEAR PIPELINES ON MULTI-CORES, CLUSTERS AND GPUS

This Master Thesis has been submitted in partial fulfillment of the requirements for the degree of Master in Computer Science of the Computer Science Graduate Program, School of Technology of the Pontifical Catholic University of Rio Grande do Sul

Sanctioned on March 20, 2025.

COMMITTEE MEMBERS:

Prof. Dr. Cesar Augusto Fonticielha De Rose (PPGCC/PUCRS)

Prof. Dr. Marco Danelutto (Univ. of Pisa)

Prof. Dr. Dalvan Jair Griebler (PPGCC/PUCRS - Advisor)

ACKNOWLEDGMENTS

This work was funded by PUCRS University, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001, CNPq Research Program (Nº 306511/2021-5), and FAPERGS call 09/2023 - PqG.

UMA DSL ALTO NÍVEL EM RUST PARA EXPRESSAR *PIPELINES* LINEARES EM *MULTI-CORES*, *CLUSTERS* E GPUS

RESUMO

Aplicações modernas com uso intensivo de dados exigem o processamento eficiente de *streams* contínuos de dados, frequentemente requerendo execução paralela em diversos ambientes de hardware, incluindo sistemas de memória compartilhada, *clusters* distribuídos e GPUs. Programar esses ambientes de forma eficaz apresenta desafios significativos devido às suas características e modelos de programação distintos. Este trabalho apresenta uma nova linguagem de domínio específico de alto nível, incorporada em Rust, para expressar paralelismo de *pipeline* linear direcionado a arquiteturas de multicore, *Clusters*, e GPU. Nós a chamamos de SPar-Rust, que simplifica a expressividade do paralelismo de *pipeline* linear e fornece uma interface de programação unificada direcionada a diferentes arquiteturas paralelas para o desenvolvimento produtivo de aplicações paralelas de processamento de fluxo. SPar-Rust realiza transformações de código fonte para fonte, onde os detalhes de baixo nível são ocultados dos desenvolvedores, usando o poderoso sistema de macros do Rust. Analisamos as capacidades do conjunto de ferramentas de Rust para geração e abstração de código, focando nas implicações de desempenho e programabilidade de nossa abordagem em comparação com soluções do estado da arte, mostrando sua eficácia e limitações. Até onde sabemos, SPar-Rust é a primeira DSL em Rust a oferecer suporte integrado para execução em GPU e uma interface de programação unificada direcionada a diferentes arquiteturas paralelas.

Palavras-Chave: Rust, Processamento de *Stream*, SPar, Transformação de Código, Paralelismo.

A HIGH-LEVEL DSL IN RUST FOR EXPRESSING LINEAR PIPELINES ON MULTI-CORES, CLUSTERS AND GPUS

ABSTRACT

Modern data-intensive applications demand efficient processing of continuous data streams, often requiring parallel execution across diverse hardware environments, including shared-memory systems, distributed clusters, and GPUs. Programming these environments effectively presents significant challenges due to their distinct characteristics and programming models. This work presents a novel high-level domain-specific language embedded in Rust for expressing linear pipeline parallelism targeting Clusters, multi-core, and GPU architectures. We named it SPar-Rust, which simplifies the expressiveness of linear pipeline parallelism and provides a unified programming interface targeting different parallel architectures for developing productive parallel stream-processing applications. SPar-Rust performs source-to-source code transformations where low-level details are hidden from developers by leveraging Rust's powerful macro system. We analyze the Rust tool-chain's capabilities for code generation and abstraction, focusing on our approach's performance and programmability implications compared to state-of-the-art solutions, showcasing its effectiveness and limitations. To the best of our knowledge, SPar-Rust is the first DSL in Rust to offer integrated support for GPU execution and a unified programming interface targeting different parallel architectures.

Keywords: Rust, Stream Processing, SPar, Code Transformation, Parallelism.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

ACM – Association for Computer Machinery

API – Application Programming Interface

AST – Abstract Syntax Tree

CPU – Central Processing Unit

DSL – Domain Specific Language

GPS – Global Positioning System

GPU – Graphics Processing Unit

GPGPU – General Purpose Graphics Processing Unit

HPC – High-Performance Computing

IEEE – Institute of Electrical and Electronics Engineers

IR – Intermediate Representation

ISO – International Organization for Standardization

JSON – JavaScript Object Notation

MPI – Message-Passing Interface

NO-OP – No Operation

PGO – Profile-Guided Optimizations

RAM – Random Access Memory

RFC – Request for Comments

SIMD – Single Instruction, Multiple Data

SIMT – Single Instruction, Multiple Threads

SLOC – Significant Lines of Code

STL – Standard Template Library

TBB – Thread Building Blocks

UB – Undefined Behavior

XOR – Exclusive 0R

CONTENTS

1. INTRODUCTION

The Bull Gamma 60, first shipped in 1960, was the world's first multi-threaded computer [10]. Throughout the 60s, great research effort was spent on increasing parallelism within the processor with execution pipelines and replication of function units [20], as this was seen as the only realistic way of increasing performance, since single core processor units had begun yielding diminishing returns [3]. In the early 1970s, the advent of microprocessors further increased our ability to connect multiple processing units into a single CPU, and in the 1980s, we already had several approaches to dealing with parallelism [20]. One of these was through shared-memory, which would eventually lead the way to the modern processor units we have today. Another was the message-passing paradigm, that we use today to program clusters of many computers.

Since its inception, programming parallel systems of all kinds has been challenging. This difficulty has manifested in several different ways, for example, the many specialized tools invented to exploit different kinds of parallelism over the years: the Erlang programming language [101] in 1986, designed to handle millions of concurrent processes[1]; the more modern languages Go [26] and Elixir [39] that serve similar purposes; the Message Passing Interface (MPI) [75] that we use to program distributed systems; OpenMP [22] that can be used to parallelize loops in C/C++ and Fortran in shared-memory environments; and countless libraries for every modern programming language in industrial use, each often focusing in one specific parallel paradigm. The difficulty is also evident in how researchers struggle to formalize modern parallel programming, because of how few guarantees modern processor architectures offer the programmer when it comes to instruction ordering and atomicity [99]. All of this is exacerbated further by the fact each parallel architecture has its own set of particular considerations and algorithms that work on them [8]: a shared-memory environment can make use of efficient semaphores and mutexes as locking mechanisms to synchronize its computation while a distributed system can only communicate through messages. These differences make it possible for one to be an expert in parallelism in shared-memory, but know very little about it in the context of distributed systems, and vice-versa.

There has been substantial research effort in the past to simplify and ease the programming of parallel systems. For example, patterns for structured parallel programming [23, 74] ameliorate the difficulty by presenting known-to-work parallel procedures with higher-level descriptions (pipeline, map-reduce, and so on). However, the programmer still must be aware of each platform's specific characteristics and programming models to be able to use them efficiently. The C++ language community has also dedicated

[1] Erlang has its own definition of a "process". It is similar to a concept we today sometimes refer to as "green threads": a small structure containing tasks to be executed, that will be scheduled by Erlang's own runtime.

much of its time to providing high-level parallel programming abstractions. Parallel patterns are instantiated via C++-based template libraries with classes and predefined functions. Examples of such abstractions are Thread Building Blocks (TBB) [79], FastFlow [2], GrPPI [25] and SkePU [29]. More recently, we can cite Kokkos [98], HPX [58] and even the Standard Template Library (STL)'s parallel algorithms [66]. All of them are trying to make it easier to write parallel programming. Furthermore, many of these frameworks worry greatly about the portability of their programs to different parallel architectures, and they can often be executed in multi-threaded and distributed systems, or even in the GPU.

Today, we live in a time where data is abundant, being produced at ever-increasing rates [18]. Raw data is often not very useful, so there is ample need to somehow process that data. Achieving this without parallelism is nearly impossible, and thus modern data-processing systems tend to be made either with very powerful multi-threaded processors, or, more commonly, many less powerful computers connected in a distributed system. Recently, specialized hardware (Graphical Processing Units - GPUs, or other accelerators) has also become more common, as these can achieve significant speedups in routines for which they are suited [47]. Specialized hardware can be even more challenging to program than more orthodox parallelism, since it involves programming for a hardware with a completely different architecture, often with special concepts and considerations that one would ordinarily not concern themselves with.

There are cases where data has become so large, is produced at such high rates, and must be processed so fast that we began processing it as soon as it arrives at the destination or reaches a time constraint. For example, to analyze comments on a social media website, we can not simply wait for everyone to stop commenting, and only then collect the data to analyze it. The analysis must be done while the data is being generated, and continuously updated as more of it arrives. Thus, the field of stream-processing was born [7].

Stream-processing can be done using any of the three aforementioned parallelism architectures: in one powerful computer with shared-memory, in many distributed computers, and using specialized hardware like GPUs. Accordingly, it can be very challenging to build correct, efficient stream-processing applications, as they can combine the difficulties of all parallel architectures at the same time. As a result, many frameworks and libraries have been created over the years to aid developers in this task.

Of the many stream-processing frameworks that came to be over the years, we can mention Storm[55] and Flink[35], as two popular frameworks for the Java programming language [9]. Java became an attractive language for stream-processing because of its portability, which allows Storm and Flink to run in highly heterogeneous clusters without many deep, low-level, programming considerations, as those are mostly handled by the Java Virtual Machine. Storm and Flink are frameworks specialized in stream-processing,

and therefore expect their users to know stream-processing concepts such as processing windows, watermarks, sources, sinks, stateful operations, and so on.

Java executes in a virtual machine, and enormous research effort has been put into making it as fast as possible [72]. Nevertheless, the Java virtual machine still can not beat ahead-of-time optimizing compilers, particularly of old and well-established languages like C++ [37]. And so, we began building stream-processing applications in C++, using the previously mentioned abstractions and frameworks. Like Storm and Flink, these frameworks often demand quite a lot from their users: for example, one must learn Kokkos' own programming model before one can begin using it efficiently. There are projects that, in contrast, aim to make stream-processing easier. SPar [40, 42] was created with this explicit purpose. It is a domain-specific language (DSL) aimed at simplifying and making the development of parallel stream-processing applications productive, as demonstrated in experiments and analysis regarding programmability[4, 5, 6]. It is based on C++11 annotation mechanisms (we explore this further in Section 2.6), which does not have standard compiler support: compilers simply ignore attributes they do not recognize[2]. And so, to make SPar work, the authors had to implement their own simplified C++11 compiler to process their annotations. C++ is a complex language[3]. As a result, SPar remains locked to C++14 to this day, as updating its language support beyond that would be a monumental task.

Throughout its versions and revisions, C++ has accumulated much historical baggage. Its international standard [54] contains 2104 pages, and it is still *under-specified*, as many of the language's semantics are described in natural language, since formalizing them in any way can often be very challenging. Macro expansion and evaluation are ambiguous and may vary from compiler to compiler. Many modern, low-level programming languages were designed without the backward compatibility constraints of C++, which may be more suited for the needs of modern applications. In this work, we will be focusing on Rust.

Rust is a relatively new programming language that has been growing in popularity. One of the hallmarks of its growth and success was its inclusion in the Linux Kernel [19]. Rust is a low-level language with a minimal runtime without garbage collection and was specifically designed as a modern and safer alternative to C/C++ [64]. It is already being used in production systems, the Linux Kernel being perhaps the most high-profile example. Rust's commitment to safety, through the explicit lack of undefined behavior, makes it an attractive option for stream processing since these applications are meant to have very long uptimes, something that can be challenging to achieve in languages with a lot of undefined behavior like C/C++, because these can lead to very unpredictable results (see Section 2.2.1). Rust also offers several modern facilities that C/C++ lack: a

[2] more specifically, the `gcc` compiler will ignore them, while `clang` will output an error.

[3] in fact, the C++ template system itself is Turing Complete [100].

modern, easy-to-use build system, well-specified code-transformations through procedural macros, a strong type system enforcing correctness, and so on. Rust's age, compared to languages like C, C++, and Java, explains why it still does not have a well-established stream-processing framework. Nevertheless, as we will see in Chapter 3, Rust already has many libraries and frameworks for all kinds of parallel architectures: multi-threaded, distributed through MPI and even GPUs. This shows the programming community believes in Rust's ability to produce competitive, highly efficient code. We believe Rust will let us offer a more portable and safe alternative to solving problems previously tackled by C/C++, while also not sacrificing performance.

1.1 Research Contributions

In this work, we create SPar-Rust, a DSL for writing stream-processing applications embedded in Rust. We support all 3 parallel architectures we discussed above, just like the original SPar, while exposing the same, unified programming API to the developer. This is done leveraging Rust's code-transformation features, which we discuss in Section 2.2.2. More objectively, our main contributions are:

- A novel, Rust-embedded, and high-level Domain Specific Language for stream-processing using macros to perform source-to-source code transformations, heavily based on the original SPar.
- Code generation algorithms for shared-memory, distributed-memory and GPU architectures. To the best of our knowledge, ours is the first work to support GPUs in a high-level manner for stream-processing in the Rust programming language.
- A unified linear pipeline abstraction and runtime that supports all three parallel architectures.
- Analysis of the Rust tool-chain for code generation and abstractions, focusing on macro transformations and their performance and programmability implications.
- A quantitative analysis composed of 7 applications. We measure their performance and programmability, comparing the results produced by our work with state-of-the-art solutions.

These research contributions were possible also due to our previous efforts in shared-memory [31] and distributed architectures [30]. In [31], we provided annotations similar to those of SPar when targeting shared-memory architectures. Afterward, in [30], we reshaped our DSL to increase abstractions while targeting only distributed architecture. In this master's thesis, we took advantage of this past experience and redesigned

the language and runtime system so that with a single way of annotating linear pipelines, we can target parallel code for multi-core, cluster, or GPUs without refactoring the source code.

1.2 Outline and Contents

We begin in Chapter 2 by presenting the required background knowledge to understand this work. We will explore steam-processing, Rust, parallel programming in shared memory, distributed and heterogeneous environments, SPar and some of our previous work. Then, in Chapter 3, we discuss related works, comparing them to what we are attempting in this Thesis. Chapter 4 explains our high-level abstraction: SPar-Rust, detailing its implementation for all parallel environments. Chapter 5 then presents SPar-Rust's experimental results, comparing it to many of the works we discuss in Chapter 3. Finally, we end with our conclusions in Chapter 6.

2. BACKGROUND

In this Chapter, we present the main concepts necessary to understand our work. These include both theoretical and practical concerns and will be used in the following chapters as a way of evaluating previous work and justifying our own. We begin by discussing stream-processing applications in Section 2.1. Then, we introduce the Rust programming language in Section 2.2, discussing its main differences and advantages compared to more traditional languages used in HPC, such as C and C++, as well as certain idiosyncrasies that will be relevant when we present our work's implementation. The three following Sections (2.3 to 2.5) briefly explain the three parallel environments we are interested in: shared-memory, distributed and heterogeneous hardware, respectively. Section 2.6 is an introduction to SPar, a DSL focused on high-level abstractions for stream parallelism, and, finally, Section 2.7 finishes with a discussion of our previous work.

2.1 Stream Processing Applications

Stream processing applications are characterized primarily by their need to process a continuous stream of data, uninterrupted, typically for a long amount of time [7]. The data can be produced through physical devices such as cameras and sensors, or large-scale applications, such as social networks. Processing it involves applying a series of operators, which can be filters or transformations, possibly storing its results [103], or simply presenting them in a way to enable real-time analysis. This often leads to high, strict performance requirements for these applications, which are commonly implemented using high degrees of parallelism to comply with these requirements [94]. Furthermore, these applications have high availability requirements, since spurious shutdowns and crashes could cost companies millions of dollars (if it is related, for example, to stock market speculation, or targeted advertisement), or worse, in the case of systems monitoring data from sensors to determine the general safety of an environment. Stream-processing applications can consume structured or unstructured data[7, 84]. Examples of structured data include JSON (JavaScript Object Notation) or database-style records. Unstructured data is far more common; these could be the raw data emitted from sensors, digital media in audio, image, and video formats, or arbitrary user text input (as may happen in social networks). Current advances in deep learning further enhance our capacity to deal with unstructured data, and indeed, artificial neural networks are one way of coping with the ever-increasing amount of data to be processed [105]. Figure 2.1 shows some examples of stream-processing applications.

The nature of these applications and the way the data they process is created have the consequence of creating an unknown input volume. They will simply execute

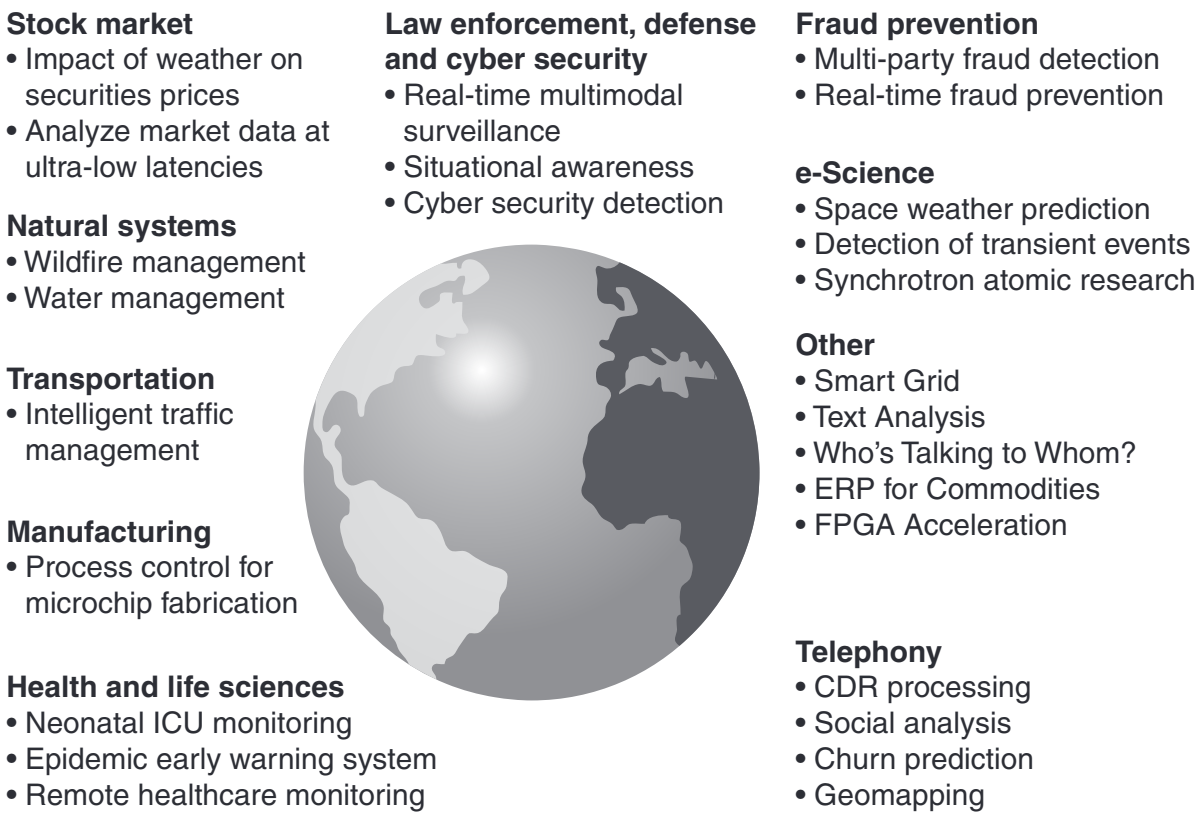


Figure 2.1: Examples of Stream Processing Applications. Extracted from [7].

continuously as more and more data arrive until they are manually shut down by their maintainers. There are two main patterns commonly used in stream-processing: linear pipelines [73] and DataFlow [1]. This work will propose an abstraction (in Chapter 4) that works particularly for modeling linear pipelines, but it is incapable of modeling DataFlow patterns. We will now present the linear pipeline pattern, followed by a brief description of the DataFlow and how they differ from each other.

2.1.1 Linear Pipelines

A pipeline is a directed acyclic graph, where its edges represent the flow of the application's data processing, and each node stands for a particular computation that consumes the data generated in the previous stage, producing the input for the next stage. Figure 2.2 shows a simple 5-stage pipeline. In it, stage 0 must receive, collect, or create the data to be processed, and stage 4 must produce the pipeline's final output. In the context of stream processing, these stages are often called *source* and *sink*, respectively.

A more realistic pipeline for production environments is shown in Figure 2.3. In Figure 2.2, we only exploit parallelism within the pipeline itself. For example, if in_1 and in_2 arrived one after the other in the source, when in_1 were in stage 2, in_2 would be in stage 1, both executing at the same time. However, the pipeline pattern lets us explore much

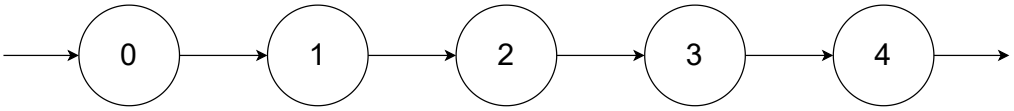


Figure 2.2: A simple 5 stage pipeline

more parallelism than that; because each stage only depends on the input created in the previous stage, we can replicate the stages themselves to increase the degree of parallel processing in our applications. This is visualized in 2.3.

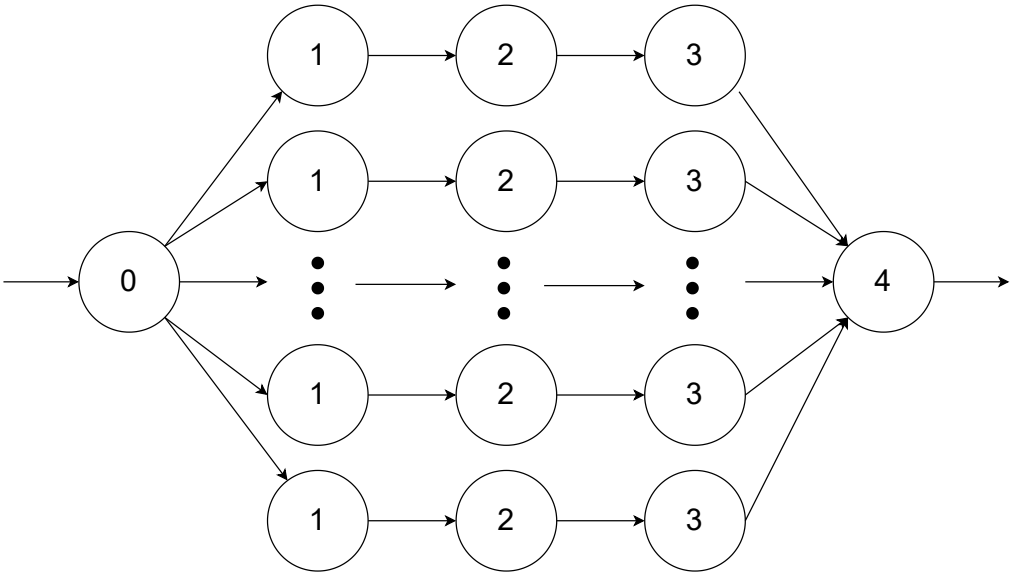


Figure 2.3: A more complex 5-stage pipeline, with parallelism within the stages.

2.1.2 Data Flow

The fundamental difference between a linear pipeline and Data Flow based stream-processing is that the latter does not need to be neither linear nor acyclic. Figure 2.4 shows an example. Note how the source can route the data to multiple nodes: *A* and *B*, each with different roles. Furthermore, the graph has two sinks: nodes *E* and *F*. This could correspond to the processing graph deciding to write data to a database, or just sending it through the network to a different system, for further processing. The pipeline is thus no longer linear: different data can take different paths through it. Finally, node *F* may decide instead to feed its data to node *G*, which would do some transformation and feed it back into node *D* for re-processing, thus creating a cycle in the graph. In these systems, the focus is no longer on stages and their connections, but on the flow of data in the system. Systems following the Data Flow pattern usually also have robust ways of recovering from failure states, being able to restart computation automatically by themselves.

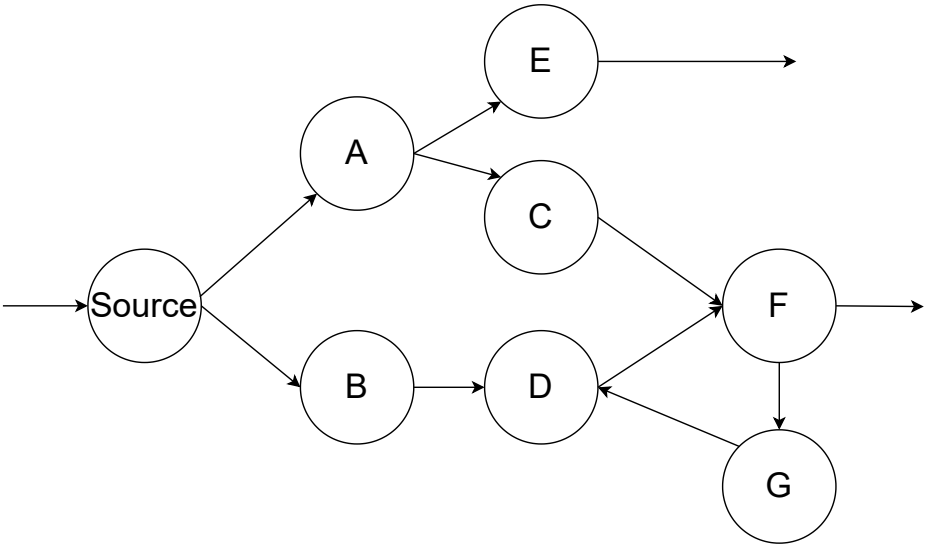


Figure 2.4: An example of a Data Flow processing graph.

Actual systems used in production tend towards the Data Flow pattern, because they have complex dependencies between stages, and complex algorithms to ensure exactly-once semantics [35], due to the mentioned reliability requirements of these applications. These semantics will often rely on writing transactions in a database, in order to be able to restart the stream processing should the system fail, with the exact same results as if it had not. As mentioned, in this work, we do not concern ourselves with these highly complicated systems. That would entail implementing an entire stream processing framework, such as Storm [55] or Flink [35], and it is out-of-scope for our work. Instead, we will focus on linear pipelines at the level of complexity of Figure 2.3.

2.2 The Rust Programming Language

In this section, we strive primarily to present Rust's advantages as a language and justify our choice of it as a low-level language that is appropriate for stream-processing applications. We will also briefly explain some of its constructs and features that are necessary to understand this work.

The Rust programming language historically came to be as a low-level alternative to C and C++ developers that did not allow for Undefined Behavior (UB) [64]. More precisely, Rust guarantees that, as long as the keyword `unsafe` is not used by the program, if it compiles, then it does not have UB. Rust programmers refer to this property as *soundness*, and code that does not contain UB is considered *sound* code.

As the Rust project has always had the goal of serving as a C and C++ alternative, the language does not have either a garbage collector nor a runtime, being able

to run even in embedded devices. Currently, Rust uses LLVM [67] as its primary code generation back-end[1], which allows it to create binaries for the same targets as `clang`, LLVM's `C` compiler. By using the same code generation back-end, Rust also ensures its performance is comparable to C/C++, and works such as [14, 34] confirm Rust's ability to deliver performance on par with C/C++ in supercomputing applications.

At the language level, Rust's main advantage over C/C++ is its safety guarantees. Rust has an intimate relationship with Undefined Behavior: it introduced a keyword just to interact with UB and was designed specifically to let the programmer do as much as possible with resorting to constructs that might cause UB. Therefore, to truly understand what Rust offers in comparison to C and C++, as well as some of its limitations, it is necessary to have a firm grasp on what exactly UB entails.

2.2.1 Safety and Undefined Behavior

Undefined Behavior comes in many forms: in programming languages, compilers intermediate representations (IR), or even hardware platforms [68]. We are primarily interested in UB defined in programming languages and secondarily in compilers' IRs.

The C programming language specification defines many kinds of UB [53]. Their original purpose was to allow C to be ported and optimized to many different architectures, where certain behaviors may differ significantly [104]. For example, several computer architectures have different solutions for integer division by zero: x86 causes a hardware exception, while PowerPC silently ignores it [104]. By setting the behavior of integer division by zero as "undefined", the C specification allows it to be easily implementable in both these architectures. C++, being a superset of C, inherited all of C's UBs, and introduced some more of its own.

Because UB means that the hardware can do anything, compiler writers have begun assuming that UB never happens when writing optimizations, allowing them to assume certain invalid states also never happen, and putting the burden of ensuring that on the language's programmers [68]. Since compilers explicitly assume UB never occurs, an unfortunate consequence is that **any resulting binary is correct if the source code contains undefined behavior**. In practice, due to a compiler's nature, it is very unlikely that the resulting program will be significantly different than what is specified in code; often UB manifests as strange bugs that occur in seemingly unrelated parts of the code, in a program that otherwise behaves as expected. Indeed, UB can sometimes cause seemingly "impossible" results, as the following example demonstrates.

[1] It is possible to use different back-ends, such as Cranelift [16], which could be used for debug builds.

Example of Undefined Behavior

This example has been largely taken and adapted from [56]. Listing 2.1, below, contains undefined behavior. Specifically, the variable x is being read without being initialized:

```c
#include <stdio.h>

int always_true(int x) {
    return x < 100 || x == 100 || x > 100;
}

int main(void) {
    int x;
    printf("%s\n", always_true(x) ? "true" : "false");
    return 0;
}
```

Listing 2.1: Example of Undefined Behavior in C

Trivially, the function `always_true` should always return true, since any number is either smaller than 100, equal to 100, or bigger than 100. However, depending on the compiler's version and the level of optimization used, this function *could* return false. Indeed, the equivalent (unsafe) Rust code *did* return false in certain compiler versions [56].

This is possible due to several complex interactions that happen during the compilation process. Uninitialized variables are UB, therefore the compiler may assume they do not exist. Whenever the compiler encounters one such variable, it keeps track of the fact it is uninitialized. Whenever our code attempts to read this variable, then, the compiler can simply choose *any arbitrary register*. Since the variable is uninitialized, it can, by definition, contain any value, and because this is UB, the compiler does not have to assume this value is the same every time the variable is read. So, for Listing 2.1 to print "false", the compiler just has to pretend that x first has any value above 100, and then any value below 100. It can then verify this always returns false and constant-fold the function to simply:

```c
int always_true(int x) {
    return 0;
}
```

Listing 2.2: Possible result of optimizing with UB

This is just one example of how UB can lead to very unexpected results when compiling code. There are many others in C/C++: use-after-free, access-out-of-bounds, dereferencing a null pointer, signed integer overflow, misaligned pointer-casting, and so on. UB can lead to bugs that are very hard to find because they result in wildly unpredictable behavior, and may cause systems to crash well after they've been executing for a long time, as well as crash only in some executions, but not others [68]. This is undesirable for stream processing applications because of the reliability requirements discussed in the

previous Section. Furthermore, many of these cases of UB can happen frequently, even in security-sensitive code: in [12], a group of researchers found that 37.2% of vulnerabilities in cryptographic libraries written in C/C++ were memory related, many of which wouldn't happen in Rust, since `null` pointers do not exist in safe Rust, and access-out-of-bounds would result in a direct crash, not leaking any information to an attacker, or allowing for remote code execution by reading beyond the buffer limit. This is the biggest advantage Rust offers over traditional HPC languages like C/C++. This entire category of bugs, which could theoretically produce any arbitrary runtime behavior, and that could, historically, only be avoided by the programmers themselves, is now detected and guarded against statically, by the Rust compiler.

The `unsafe` keyword

The `unsafe` keyword essentially creates two versions of the Rust language: *safe Rust* and *unsafe Rust*. Safe Rust is *sound* by default — any UB in safe Rust is immediately considered a bug by the language maintainers. Unsafe Rust is *sound* if and only if it does not contain UB, and the programmer must manually ensure that certain invariants are upheld[2]. Generally, when programming in Rust, it is expected to only use `unsafe` when absolutely necessary, documenting precisely the conditions under which running that code is sound, and ensuring they are valid every time it is used. In this work, whenever there is no explicit mention of unsafety, it is assumed we are speaking of safe Rust.

2.2.2 Code Transformations in Rust

Rust, as a programming language, has another advantage over C/C++ when it comes to our work: meta-programming and code transformation. Rust offers us standardized ways of doing things that in C/C++ could only be done, non-portably, either through non-standardized preprocessor directives, or compiler plugins (or by writing a compiler yourself). Meta-programming in Rust is done through *macros*, which differ significantly from their counterparts in C/C++. The most important part being that Rust integrates the macro system into the language itself [92], clearly specifying its behavior, rather than leaving some of it to an implementation-defined preprocessor.

Macros in Rust come in 2 forms: *declarative* and *procedural*. We will examine each of them in turn.

[2] Checking unsafe Rust for UB automatically is an active area of research (for example, the Stacked Borrows aliasing model [57]). The Rust Project offers Miri [76] as a tool that can help with that. Though it can not detect all cases of UB and may also report some false positives.

Declarative Macros

Declarative macros perform advanced text substitution on the language tokens they receive as input. The programmer specifies exactly what kind of token can be passed as input (for example, an identifier, a type, or an expression), and the macro will substitute the calling code by its body, performing the specified expansions when necessary.

These can only interfere with the code outside of them in limited ways. For example, they can only set a variable's value if the variable was passed to it as an argument when it was called or if the variable was declared at the macro's definition site. In contrast, the C/C++ preprocessor will simply perform the substitution every time, even when the code modifies the surrounding environment in arbitrary ways. Rust calls macro that do not modify their surrounding environment "hygienic" [92]. It is considered good practice for declarative macros to be hygienic, since that makes their behavior more predictable for the programmer. In Chapter 4, when we present our `to_stream` declarative macro, we precisely indicate in which cases it is not hygienic. Note that even unhygienic macros must still generated code that conforms to Rust's safety guarantees, since they merely generate code that will later be compiled normally.

Procedural Macros

Rust macros have advanced capabilities beyond text manipulation. Specifically, *procedural* macros are implemented as user-defined programs that can receive a stream of Rust tokens that they can analyze and transform, outputting a stream of tokens that reflects the desired transformation. During compilation, when one of these macros is invoked, the compiler will execute the respective program and feed the parsed Rust tokens as input, expecting a stream of valid Rust tokens as output. Within the procedural macro, the developer can do any arbitrary computation. It is possible to, for example, generate code conforming to a specification contained in a separate file, detect special hardware features, adjust the generated code accordingly, and so on.

There are 3 types of procedural macros according to the Rust Reference [92]:

1. *Derive macros* must precede `structs` or `enums` to generate code that implements certain functionalities;
2. *Attribute macros* can define custom attributes on any item;
3. *Function-like macros* are similar to functions but operate on their arguments as tokens, not values.

In this work, we use attribute macros to perform function transformations. When used in this way, attribute macros look very similar to annotations in other programming

languages such as C++ and Java. Listing 2.3 shows an example. Here, the Rust compiler will feed the parsed tokens of `function_to_transform` (from the `fn` keyword to the last closing curly-bracket of the function's body) to a program called `example_macro`, that will then output the transformed Rust code.

```
#[example_macro]
fn function_to_transform(
    /*inputs*/
) -> /*outputs*/ {
    /* function body */
}
```

Listing 2.3: Attribute macro example

2.2.3 Traits and Generics

Traits are Rust's names for what other languages sometimes call *interface*: a contract that says a certain type implements certain methods. A type that implements the `Display` trait, for example, can be formatted and printed to standard output by using the `println` procedural macro, because it has a `Display::fmt` method. Traits let the programmer write generic code that accepts any type that implements a certain trait. For instance, a function with the signature "`fn f<T:Display>(x:T)`" will accept as input any type that implements the `Display` trait. Then, at compilation time, the Rust compiler will monomorphize the function — essentially, create a different version of the function for every necessary type.

The following Section will introduce several traits that are important to our work. We will also define many traits ourselves in Chapter 4.

2.2.4 Threads, References and Lifetimes

To end this Section, we will be examining what Rust offers as a language in terms of parallelism. Parallelism is necessary to achieve acceptable performance in any serious instance of stream-processing. We deal with 3 types of parallelism in our work: multi-threaded based on *threads*, distributed computing based on *MPI* [75] and GPU based on OpenCL [60]. We explain each of these in turn in the following Sections. For now, we turn our attentions to Rust's support of multi-threaded programs, and the challenges its safety rules can impose on the programmer. For the purposes of this Section, it suffices to understand that a *thread* is an independent sequence of instructions that will be scheduled for execution by the operating system [91]. A single process (a program), has at least one thread, and can have as many as the operating system will allow it. A program with multiple threads is aptly named: multi-threaded.

The Rust standard library offers ways of spawning and synchronizing threads. There are abstraction for mutexes, conditional variables, atomic operations, channels, read-write locks, and barriers. Using these primitives can be challenging, because they must also conform to Rust's safety rules [38]. There are two fundamental rules to Rust's borrowing system:

1. there can be an arbitrary number of immutable references to the same data; XOR
2. there can be one, and only one, mutable reference to a piece of data

In Rust, all references must always be valid. In other words, they must not be null and the value they are referring to must be in a valid state (for example, a boolean value must be represented by a 0 or a 1. If it is a 2, then its state is invalid, and a reference to it is also invalid)[3]. Furthermore, the value must *exist*. This is non-trivial if the value refers to e.g. a Vec, the standard library vector type, because it uses heap-allocated memory. When Vec goes out of scope, it will run its destructor, and all references that point to it would be considered invalid. Rust's solution to this problem is the concept of *lifetimes*. Essentially, the Rust compiler tracks the time during which a reference is valid, and will refuse to compile code where a reference outlives an object's lifetime. Listing 2.4 shows a simple example illustrating the concept[4].

```
let v1 = Vec::new();
{
    // This reference is valid, because v1 will be valid
    // for all times while this reference exists.
    let v1_reference = &v1;
} // v1_reference lifetime ends here.
{
    // This reference is valid *right now*.
    let v1_reference = &v1;
    // This moves the 'v1' vector into the function.
    // The v1_reference is now invalid, because the
    // function will run v1 destructor at some point.
    function(v1);
    // **this is a compiler error!**. We are trying to
    // read from an invalid reference.
    let x = v1_reference[0];
} // v1_reference lifetime would end here, but
  // v1 is no longer valid because it was moved into
  // `function`. This causes the compilation to fail.
```

Listing 2.4: Lifetime example

This leads to a problem when writing multi-threaded code: a thread can be alive for the entire duration of the program. This implies any references it uses must also be

[3]Note this is true of both safe and unsafe Rust. A common programming mistake in unsafe Rust is creating a temporary invalid reference. Its mere existence is considered, itself, UB. Of course, in safe Rust, it is impossible to create an invalid reference in the first place.

[4]Rust has a rich vocabulary to talk about references and lifetimes: borrowing values, dropping values, validity, etc. We are simplifying these concepts to focus on the specific parts that will be most relevant to our work.

alive while the program is running. Indeed, looking at the standard library's implementation of `thread::spawn`, it demands the values we pass to it be `'static`, which is a special lifetime indicating the value will exist during all of the program's execution. Alternatively, instead of passing references, one could simply clone the value, though this would inevitably incur in performance sacrifices. This was eventually considered to be too limiting, and, in Rust version 1.63.0, a different API was developed: `thread::scope`. Scoping allows the user to spawn threads with references that aren't `'static`, by ensuring that all threads will be waited on at the scope's end. Listing 2.5 shows how the thread scope API works.

```
std::thread::scope(|s| {
    // spawn 1 thread
    s.spawn(...);

    // spawn 2 threads
    s.spawn(...);

    // here, at the scope's end, Rust will wait for all spawned
    // threads to finish executing. Thus, any reference that
    // outlives the scope is allowed to be used in the threads
});
```

Listing 2.5: Thread Scope

This is important because, in Chapter 5, we will examine multiple Rust multi-threaded libraries/frameworks. These will offer us their own APIs to spawn units of work in parallel. If they expose an API similar to `thread::spawn`, we will run into lifetime problems. To solve many of these problems, without resorting to cloning values (which would negate a lot of the performance gains), we will have to make use of `unsafe` to extend the duration of the lifetimes[5]. In contrast, if they offer something similar to `thread::scope`, we will be able to use idiomatic, fully-safe Rust. As one of Rust's main selling points is its safety, APIs that allow us to write fully-safe Rust and still retain performance should be considered superior to those that do not.

Finally, note that, because of the second borrowing rule, presented above, `thread::scope` is not helpful when we need *mutable* references, because only one thread would ever be able to receive that reference. This will also come to play in Chapter 5, where we will have to resort to using raw pointers, writing to them with `unsafe`.

2.2.5 The `serde` library

When executing code in parallel in a distributed environment, we can no longer pass references to data, because the address spaces of each process in each machine will be different. As such, we will have to copy the data between each computer, com-

[5] This is one of the most advanced and dangerous things one can do with `unsafe`[93]

municating through a network. These computers may not necessarily have the same configuration, or even architecture, and so, the data must be serialized into a normalized format, so that when the receiver gets the data, it can de-serialize it into a format it can understand.

Serde[89] is a serialization library widely used in the Rust ecosystem. It handles trivial cases of serialization through derive macros (a type of procedural macro, see Section 2.2.2), and gives the primitives to manually write the (de)serialization procedures when we need to do so. The serde library is used in all code targeting a distributed environment in this work.

2.2.6 Send, Sync, Serialize and Deserialize traits

As explained above, the requirements of code that executes in a shared memory environment and in a distributed one are different. This also translates in the traits of the types we want to use in these contexts must implement.

For a multi-threaded scenario, types must implement the Send and/or Sync traits. These are marker traits (so called because they do not have any methods) that indicate that the type is safe to me moved across thread boundaries (Send), or that a reference to it can be moved across thread boundaries (Sync). These are implemented automatically for all types the Rust compiler know are safe, which is essentially all fundamental types except raw pointers, and `structs` made up of those fundamental types. `Structs` with raw pointers must be manually marked as Send and/or Sync. Because the programmer needs to manually ensure these pointers are not pointing to data that can only exist in the main thread, the Send and Sync traits are unsafe, and require using the `unsafe` keyword to implement.

For a distributed scenario, types must implement the `Serialize` and `Deserialize` traits from the `serde` library. These include methods to help `serde` transform those types to and from a serialized format. `Serde` already implements these traits for all fundamental types (except pointers and references), as well as some containers from the standard library (most noticeably, `Vec`). It can also automatically implement those traits for `structs` composed of those types, through a derive macro, as mentioned. Unlike Send and Sync, these traits are not unsafe, and can be manually implemented entirely in safe Rust.

2.3 Shared Memory Parallel Programming

As mentioned previously, our work focuses on 3 different kinds of parallel systems. The first is the shared memory model, whereby one machine has many processes

that all share a single physical memory address space [21]. In the context of shared memory, it is most common to make use of *threads*, rather than *processes*. Threads are sometimes referred to as lightweight processes [91]. In this work, we will always use *process* to refer to "heavyweight" processes, so that there is no confusion between the terms. Figure 2.5 displays the shared memory environment we are envisioning.

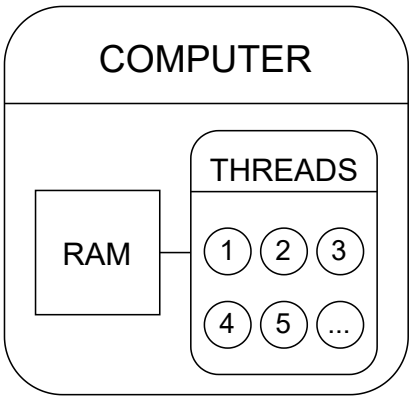


Figure 2.5: The shared memory programming model.

There are a few key differences between processes and threads [91]. Processes "own" their resources: the operating system will typically handle virtual memory addresses, file descriptor tables, and other things necessary for program execution to a process. The process can then fire as many threads as it wants by making the appropriate *syscall*[6], and will have at least one: the *main* thread. The operating system scheduler works at the thread level, not the process level. This is why Figure 2.5 has "threads", and not "processes". It is also why many works in this model, including ours, refer to it as a "multi-threaded" environment. It is possible to create parallel systems using multiple processes, rather than threads, but in this work we only concern ourselves with the later.

As Figure 2.5 shows, all threads in a machine are connected to RAM. This implies they have the same physical address space[7], which, in turn, means that they can communicate very cheaply. Accessing a pointer through one thread will yield the same address as accessing it through a different thread in the same process. Communication is, thus, virtually instantaneous; the problem is synchronization. Synchronizing threads in a shared-memory environment is a complex topic, and we will not get into details in this Thesis. We will simply note one of the myriad of problems that can arise: if two threads try to modify the same value at the same time, data races can occur [21]. However, because Rust forbids two mutable references to the same data to exist at the same time, **data races are impossible in safe Rust**, which is a distinct advantage the language has over C/C++.

[6] In the Linux operating system, the syscall is called `clone`.
[7] Note the operating system will still associate each process with a different *virtual* address space.

2.4 Distributed Parallel Programming

The distributed programming model typically involves several machines distributed and communicating through a network, all collaborating to solve one problem [21]. Figure 2.6 depicts a distributed system with an unknown number of computers (also called nodes). Every node is a full-fledged machine, composed of the same parts as a single shared-memory system.

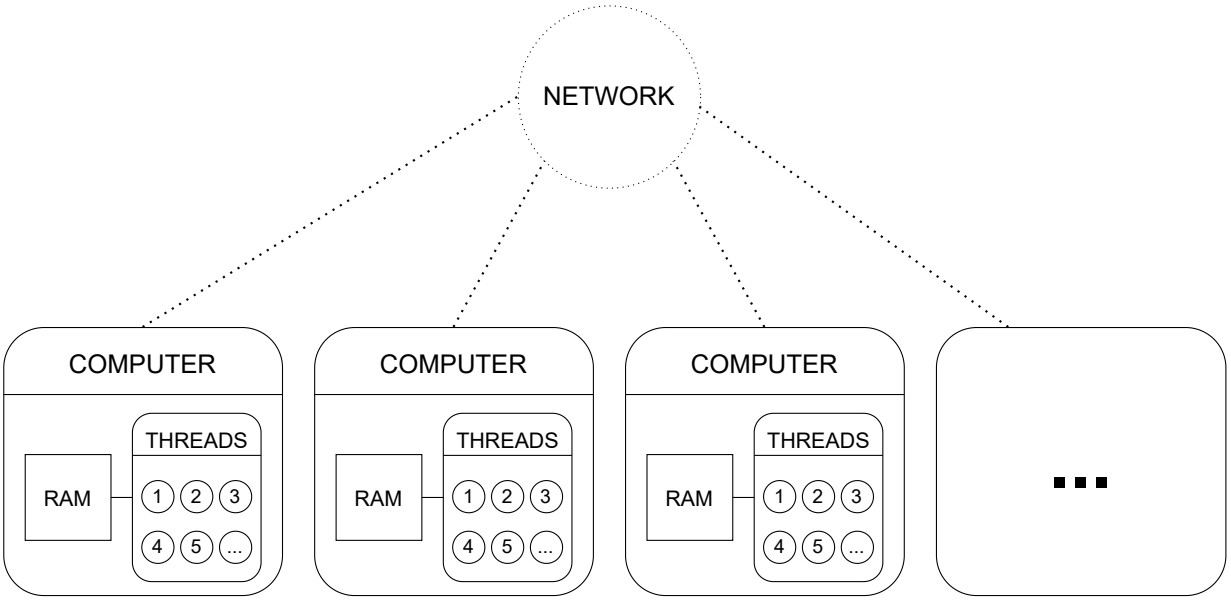


Figure 2.6: The distributed programming model.

Each machine in the system is different: their physical address space are not the same. Therefore, we can no longer communicate simply by passing pointers to address around. Instead, communication must be done through the network, by sending and receiving values through pre-established protocols. In this work, we will primarily be using MPI as our means of communication in distributed systems.

2.4.1 MPI

The Message Passing Interface (MPI) is a message-passing specification used in distributed environments to facilitate their parallel programming. MPI was developed through a coordinated effort of several researchers and organizations as a way of standardizing a message passing interface that could be used in large parallel systems [75]. Prior to MPI, these systems would have different APIs for achieving similar goals, which made writing parallel code very machine-specific and non-portable. Today, programmers

can rely on most supercomputers and large parallel system having a working MPI implementation the can use to explore their parallel computing capacity.

We will not be using any advanced features of MPI in this work, just its basic send and receive capabilities. When sending or receiving messages, MPI lets the programmer tag them with a number. We will use this tag to track the messages' order, which will be relevant in Section 4.5.4. As mentioned previously, the messages will be serialized with the `serde` library. All communication done with MPI in this work follows the same pattern of serialization→send the serialized bytes→receive the serialized bytes→deserialization.

Because of serialization considerations,

The MPI implementation we will be using in this work is OpenMPI [36]. As OpenMPI is implemented in C, we use the `rsmpi`[8] library to interact with it. `rsmpi` is a simple library that can generate bindings to OpenMPI, and offers a high-level, more rust-like interface to interact with it. It also exposes the lower level, direct C function calls, should the programmer need it.

2.5 Heterogeneous Parallel Programming

Broadly speaking, an heterogeneous system is any system that has some kind of specialized hardware to deal with specific tasks. Figure 2.7 shows a simple, general example. Heterogeneous systems can vary widely according to the kind of hardware attached to them. In this work, we focus only on Graphics Processing Units (GPUs)[9].

2.5.1 GPU Programming

GPUs have a fundamentally different architectural design from Central Processing Units (CPUs) [63]. This can be visualized in highly simplified form in Figure 2.8. From it, we can see that GPUs are organized into an array of highly threaded multiprocessors. These are organized in groups, each sharing a control unit and the cache.

Figure 2.8 is enough to show the primary advantage the GPU has over the CPU: it is inherently highly parallel, allowing for significantly more throughput than the CPU. We can also visualize some of its disadvantages: the presence of only one control and cache unit per thread group implies that the GPU will struggle with highly divergent code (code with multiple unpredictable branches) [63]. GPUs are tailor-made to handle repetitive, independent operations on large quantities of data in a SIMD (Single Instruction, Multiple

[8] https://github.com/rsmpi/rsmpi

[9] This is potentially a misnomer, since modern GPUs are not necessarily made to accelerate graphic-related tasks. Nevertheless, this is still how most of the literature calls them.

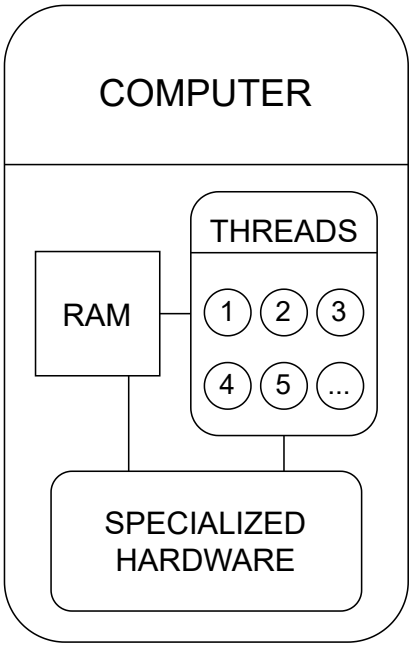


Figure 2.7: An heterogeneous system.

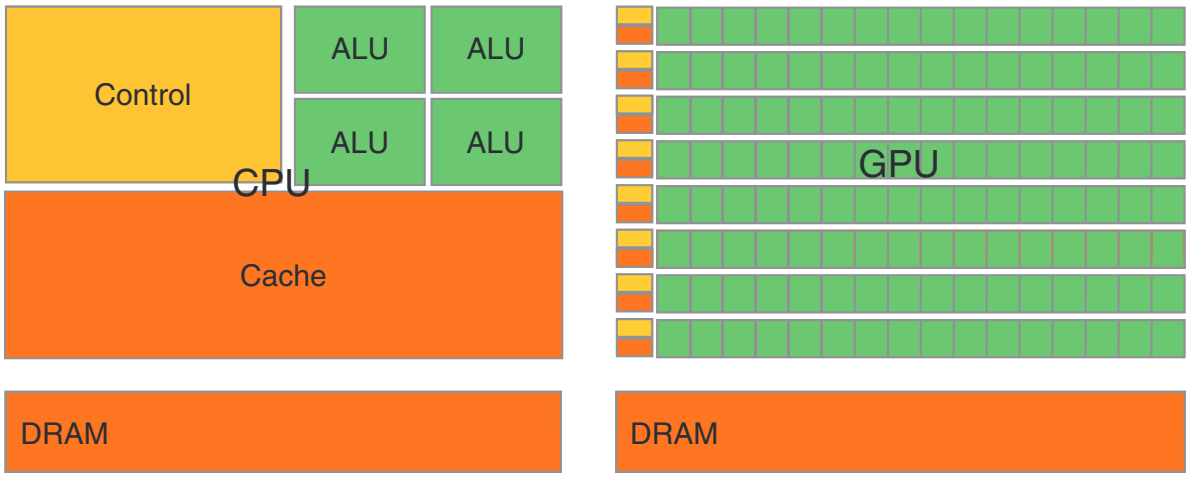


Figure 2.8: CPU vs GPU architectures. Extracted from [63].

Data) fashion. If the instructions running on the GPU contain a branching statement that goes 50% of the time one way and 50% the other, the GPU will essentially be forced to run both codes to completion twice, each time discarding half of the produced output. NVIDIA refers to this process as masking, and what it actually does is disable the divergent threads until the current ramification finishes [77].

Describing it in more detail, a modern NVIDIA GPU contains streaming processors, or CUDA cores, which are similar to a CPU's arithmetic logic units, each containing an instruction cache and sharing control units. Streaming processors are then grouped in streaming multiprocessors, which are analogous to AMD's compute units. NVIDIA calls the programming model SIMT (Single Instruction, Multiple Thread) [77], which refers to the fact that every thread in a streaming multiprocessor will execute the same instructions. The streaming processors are grouped in warps, which will then finally be scheduled for execution. Thread masking only happens at the warp level, meaning that all threads (currently a total of 32) must agree in execution flow to maximize performance [77].

Finally, we note that Franzén and Östling have showed in [34] that Rust can achieve performance comparable to that of C++ in GPU-oriented applications. However, they explain that certain adaptions to the code are necessary; one can not simply do a naive line-by-line translation of the C++ code.

2.5.2 GPGPU Programming APIs

There are two widely used general-purpose GPU programming APIs: CUDA [78] and OpenCL [60]. We will describe them briefly in this Subsection. Our code generation currently only targets OpenCL, but it is possible to extend it to also target CUDA in future works.

CUDA

CUDA describes itself as a general-purpose parallel computing platform and programming model [77]. It is specific to NVIDIA graphical cards and offers two APIs: one higher-level Runtime API, that requires the `nvcc` compiler and abstracts away tasks such as driver initialization, kernel compilation, and context management; and a lower-level API, which is a library callable from normal C/C++ code and requires the program to do everything manually themselves. As mentioned, `rust-gpu-tools` will take care of that for us.

CUDA is very similar to C, with the difference that functions to be executed on the GPU must be annotated with `__global__` (these functions are also called *kernels*), and called with a special configuration syntax «`<...>`» before its parameters. Since, for our particular use case, we will be calling these functions directly from Rust code, we only care about the syntax of their definitions. Listing 2.6 shows a simple example of defining a function that simply adds two vectors, extracted from [32]. Note the example was chosen merely for its simplicity, and is not particularly good code for the GPU, since a `for` loop will result in divergent code, and is bad for performance, for reasons previously explained.

```
extern "C" __global__ void add(uint num, GLOBAL uint *a, GLOBAL uint *b, GLOBAL uint *result) {
    for (uint i = 0; i < num; i++) {
      result[i] = a[i] + b[i];
    }
}
```

Listing 2.6: CUDA example, extracted from [32]

OpenCL

The OpenCL specification states that: “OpenCL (Open Computing Language) is an open royalty-free standard for general purpose parallel programming across CPUs, GPUs, and other processors, giving software developers portable and efficient access to the power of these heterogeneous processing platforms” [60]. One notices, therefore, that OpenCL has a larger scope than CUDA: it wishes to facilitate writing parallel code for many different platforms, not just GPUs. Furthermore, OpenCL has the advantage of not being tied to a specific vendor, since it is open and royalty-free. Despite its capabilities, we will only be using OpenCL to execute code in the GPU.

OpenCL includes a language, API, libraries, and a runtime system. Because of what `rust-gpu-tools` (explained in the next Section) offers us, we will only interact directly with its language. Much like CUDA, OpenCL is very similar to C, demanding that functions be executed on the GPU to be annotated with `__global`. Listing 2.7 shows the previous example adapted to OpenCL.

```
__global void add(uint num, GLOBAL uint *a, GLOBAL uint *b, GLOBAL uint *result) {
    for (uint i = 0; i < num; i++) {
      result[i] = a[i] + b[i];
    }
}
```

Listing 2.7: OpenCL example, extracted from [32]

There are other differences in regards to the naming and presence of certain global variables that come into play when implementing the code generation. Since we only target OpenCL for now, these differences are not important for this work. The details of our code generation will be explained later in Chapter 4.

2.5.3 The `rust-gpu-tools` library

Initializing the API’s context, moving data to and fro the GPU, as well as scheduling and executing kernels in it are operations specific to each API. In this work, we use the `rust-gpu-tools` library that abstracts that away for us, letting the programmer write very similar Rust code that will execute with CUDA or OpenCL. As mentioned, we will only

use OpenCL in this work, but this allows it to eventually be extended to support CUDA as well in future work.

In Section 2.2.6 we explained the traits that must be implemented in data we wish to use in multi-threaded and distributed environments. Here, the requirements are much more stringent. `Rust-gpu-tools` will only let us send 4byte signed and unsigned integers and vectors of fundamental integer or floating point types to the GPU. It is technically possible to encode a more complex type in a byte array, doing a sort of primitive serialization. However, de-serializing that later in the GPU would be much too costly, as it would involve complicated indexing operations, often with many conditionals, which the GPU is terrible at executing.

2.6 SPar

SPar is a Domain Specific Language (DSL) that offers high-level abstractions for stream processing making use of parallelism [40, 41, 42]. It was originally conceived for C++11 and implemented using its attributes feature and a custom compiler that processes these attributes to make source-to-source transformations on the annotated code. The language has been extended to support data parallelism [70] and service level objectives [45]. SPar's compiler was developed in subsequent works to generate code for different environments and runtimes. For shared-memory architecture, there is support for, FastFlow [42, 69], TBB [50] and OpenMP [51]. There is also support for self-adaptive stream-processing [45, 102]. For distributed architectures, there are the works of Griebler [43], Loff [71] and Pieper [82]. For the GPU there are the works of Rockenbach [84, 85, 87]. Moreover, it has shown important usability benefits presented by beginner developers [5] and coding metrics [4, 6]. Our work is, in essence, a recreation of SPar implemented in Rust, targeting the same 3 environments: multi-threaded, distributed and GPUs. This Section focuses on understanding SPar's language and basic transformations. These will later be used and adapted to implement our own code transformations in Rust.

2.6.1 SPar Language

SPar's language, though originally designed for C++11, is very generic and flexible, since it is largely based on stream-processing's basic concepts, as explained in Section 2.1. This makes it possible to adapt it and re-implement it in other languages, using their particular constructs.

The original SPar language was composed of 5 attributes:

1. `ToStream` – used to indicate the beginning of a stream processing region. This would typically be the equivalent to node 0 (the source) in Figure 2.3.
2. `Stage` – represents a *pipeline* stage.
3. `Input` – defines the input dependencies of the Stage or the stream region.
4. `Output` – defines the output of a Stage or stream region.
5. `Replicate` – specifies the number of workers in a Stage.

Note that, while `ToStream` will usually indicate the stream's source stage, *there is no attribute that stands for the sink stage*. This is because, in the original SPar's implementation such an attribute was unnecessary. SPar's compiler would figure out by itself how to write the sink stage depending on the `Output` properties to the `ToStream` region.

There were many experiments to extend the language with extra attributes [69, 87], that help improve the generated code's quality. Of particular interest to us is the `Pure` attribute, introduced in [86, 87]. `Pure` indicates the annotated code is a pure function, that is, a function whose output depends solely on its input, and does not modify any state. More specifically, `Pure` guarantees that:

1. the annotated code does not have any side effects (mutates external state).
2. the annotated code does not have execution order dependency (such as depending on values modified by previous iterations).
3. the annotated code does not access any global variable that is not listed in the `Input` attribute.

These conditions are necessary to allow for efficient GPU implementation of the underlying code. Since the GPU executes code in a massively parallel fashion, any persistent state that will be mutated by the computation must be synchronized, which is expensive and will hurt performance. Fortunately, it is possible to make our Rust API force most of these conditions by default through its type system (we explain in more detail in Section 2.7).

Furthermore, our implementation will do away with the attributes of `Input` and *Output*, since those too can be inferred from the code itself, change `ToStream` to `Source`, to explicitly stand for the source node, and add a `Sink` attribute, to indicate the end of the stream. `ToStream` will then become a declarative macro that will connect all transformations together. This will be further explained in Chapter 4.

2.6.2 SPar C++ example

Listing 2.8 shows an example of how SPar's annotations (highlighted in blue) could be used in C++ to accelerate an image processing routine. The application applies a series of filters to a sequence of images.

```
1  [[ToStream]] while( (image = nextImage()) != NULL ) {
2      [[Stage,Input(image),Output(image),Replicate(n)]] {
3          saturation(image);
4      }
5      [[Stage,Input(image),Output(image),Replicate(n)]] {
6          emboss(image);
7      }
8      [[Stage,Input(image),Output(image),Replicate(n)]] {
9          gamma(image);
10     }
11     [[Stage,Input(image),Output(image),Replicate(n)]] {
12         sharpen(image);
13     }
14     [[Stage,Input(image),Output(image),Replicate(n)]] {
15         grayscale(image);
16     }
17 }
```

Listing 2.8: Image processing in C++ with SPar

Each one of the filters in Listing 2.8 will become a stage in the pipeline, and the source will simply send the `image` returned by the `nextImage()` call to the next pipeline stage. To do this, the SPar compiler implemented complex code transformations that consumed the annotated code and produced parallel code according to the selected backend [40]. Although the transformation rules are expressed generically, in a higher-order, logical manner, they are still highly specialized to this particular version of SPar's implementation. It is analogous to how a traditional compiler's backend generates code only to a specific instruction set, and thus must be re-implemented should we want to support multiple architectures. Therefore, despite SPar's language and the general semantic meaning of its tokens being largely unchanged (for example, a `Stage` will continue to represent a `Stage`, and `Replicate` continues to indicate the degree of parallelism present in the `Stage`), we will have to rewrite and adapt the specific transformation rules for our implementation.

2.6.3 The SPar Compiler

SPar has a custom compiler that consumes code annotated with the aforementioned keywords and generated parallel C++ code in one of its implemented runtimes. The generated code can then be compiled by a regular C++ compiler. Thus, SPar com-

piles its C++ source twice: once to generate the parallel code, and another to generate the final executable.

Applying the code transformations to generate parallel code is a non-trivial task, explained in great detail in the works that implemented it for each runtime SPar support. In essence, SPar detects the parallel pattern that best models the behavior of the written sequential code, such as a *map* and/or *reduce*. It then creates an Abstract Syntax Tree (AST) representing the processing pipeline with those patterns. The AST may be transformed or simplified according to formal rules explained in [40]. Once SPars has applied its transformations, it then generates code implementing those patterns for the relevant runtime.

In contrast, our Rust implementation relies a lot more on native language features than the original SPar. This is largely because Rust offers better tools for metaprogramming and creating custom syntax than C++.

2.7 Initial Work

In [31], we presented the first version of our Rust adaptation of SPar. It was based on procedural function-like macros, and an example of its usage can be seen in Listing 2.9.

```
to_stream!({
  for image in all_images {
    let img = image;
    STAGE(INPUT(img: Image), OUTPUT(img: Image), REPLICATE = n, {
      filter::saturation(&mut img, 0.2).unwrap();
    });
    STAGE(INPUT(img: Image), OUTPUT(img: Image), REPLICATE = n, {
      filter::emboss(&mut img).unwrap();
    });
    STAGE(INPUT(img: Image), OUTPUT(img: Image), REPLICATE = n, {
      filter::gamma(&mut img, 2.0).unwrap();
    });
    STAGE(INPUT(img: Image), OUTPUT(img: Image), REPLICATE = n, {
      filter::sharpen(&mut img).unwrap();
    });
    STAGE(INPUT(img: Image), REPLICATE = n, {
      filter::grayscale(&mut img).unwrap();
    });
  }
});
```

Listing 2.9: Image processing in Rust with SPar-Rust

Our explicit goal at the time was to make it syntactically similar to SPar's C++ version we've shown in Listing 2.8. It uses nearly the same set of special tokens as the original SPar, with the main difference being most of the tokens are in uppercase, and `to_stream` is in snake case.

Since then, we've moved towards a different implementation based on Rust's procedural attribute macros, which look like function annotations in other languages. The new version of SPar-Rust was presented in [30], already with a multi-threaded and MPI backend. Listing 2.10 has an example it.

Because of Rust's borrowing rules, this new version has many useful properties: as long as it does not use `unsafe`, it *cannot mutate global state*. As long as it does not have mutable references, it *cannot mutate non-local state*. These are two of the three conditions necessary to apply the new `Pure` attribute we presented earlier. This means that, while annotating code with `Pure`, the programmer would only have to ensure that:

1. the function does not use `unsafe` (we can do this automatically).
2. the function does not receive mutable references as input (we can also do this automatically).
3. the function's code does not have execution order dependency (the only thing the programmer would have to verify manually).

```
#[source]
fn source() -> impl Iterator<Item = Image> {
    all_images.into_iter()
}
#[stage]
fn saturation_filter(img: Image) -> Image {
    filter::saturation(&mut img, 0.2).unwrap();
    img
}
#[stage]
fn emboss_filter(img: Image) -> Image {
    filter::emboss(&mut img).unwrap();
    img
}
#[stage]
fn gamma_filter(img: Image) -> Image {
    filter::gamma(&mut img, 2.0).unwrap();
    img
}
#[stage]
fn sharpen_filter(img: Image) -> Image {
    filter::sharpen(&mut img).unwrap();
    img
}
#[stage]
fn grayscale_filter(img: Image) -> Image {
    filter::grayscale(&mut img).unwrap();
    img
}
#[sink]
fn sink(img: Image) -> () {}
```

Listing 2.10: Image processing in Rust with SPar-Rust V2

This is one of the advantages of both Rust and this new version of SPar-Rust. There are others, such as simplified implementation, because the old version could access any variables previously declared in the same scope (i.e., its macro was not fully hygienic), while the new version only has to deal it variables passed to the function.

This Master Thesis is based entirely on this second version of SPar-Rust. In Chapter 4, we will present its code transformation strategies, for all runtimes it supports. We also discuss its limitations, particularly regarding the GPU implementation, which is very limited in its scope, and can only successfully transform code written in a specific way. It is worth noting that our previous work already supported both multi-threaded and MPI. However, the MPI implementation still had some limitations, the most important one being: we could only have one stream-processing pipeline per application. And so, there was a semantic mismatch between the MPI and multi-threaded implementations that could come into play when modeling more complex applications that require multiple processing pipelines. This is no longer the case, as we now offer a truly unified interface for all supported runtimes. The GPU backend does impose limitations on how one can program the functions to be transformed. However, this is a limitation of the code transformation algorithm, not of the DSL's API.

3. RELATED WORK

To the best of our knowledge, there are no works in the literature that attempt to offer a single unified interface to writing parallel code for multi-threaded, distributed and GPU environments in Rust. As such, we have selected related work for each of these runtimes and will discuss each of them in turn. Every related work presented in this Chapter should be assumed to only efficiently support one runtime, unless stated otherwise. At the end of this Chapter, we show a comparative table for an easy way to visualize the differences between all the works.

3.1 Multi-Thread

Rust's claim of "fearless concurrence" has attracted many developers and academics to create multi-threading libraries. Rayon [83] and Tokio [95] are two of the most popular libraries routinely used for concurrency in multi-threaded environments.

Rayon is implemented with a thread-pool with a work-stealing algorithm based on a double-ended queue. Its API mirrors the iterator API of Rust's standard library, with a `par_` attached to the relevant functions. For example, `into_iter()` becomes `into_par_iter()`. In the best-case scenario, this makes parallel code trivial to write: simply change a handful of function calls, including the `par_` prefix wherever necessary. However, sometimes, because of Rust's borrowing rules, the resulting code will be rejected by the compiler, which will warrant rewriting the computation logic to appease it. Cases where this happens will be presented in Chapter 5, when we discuss our experiments.

Tokio is a library based on Rust asynchronous computation capabilities. The programmer creates asynchronous task that are then executed in the Tokio runtime. The runtime takes care of scheduling these tasks in multiple threads, as it deems necessary. The model is very similar to that of Go's *goroutines* [27]. Tokio's primary use-case is creating web servers that can handle thousands of concurrent connections at once.

Besides Rayon and Tokio, there have been many academic works focused on Rust's multi-threaded capabilities.

`Rust-SSP` [80, 81], by Pieper et al., adopts a structured approach to parallelism. It provides primitives allowing the user to create their own processing pipeline, while also offering a declarative macro that simplifies that process. It has become a somewhat old project, with very little maintenance. It was one of the early academic works in developing high-level abstractions for stream processing, and has since been superseded by more modern works. As such, we do not include `Rust-SSP` in our experiments in Chapter 5.

PPL [11], by Besozzi, is essentially a more modern, feature-full Rust-SSP. It is also based on the concept of structured parallelism, supporting the creation of processing pipelines like Rust-SSP, but with a richer interface and more pre-made patterns to explore. The authors themselves compare it to Rust-SSP in their paper. PPL is one of the libraries we will compare our work to in Chapter 5.

Sydow et al. [90] took a different approach, using profile-guided optimization (PGO) to produced efficient parallel, stream-processing Rust code. PGO involves compiling the program with instrumentation, running it to collect execution data, and then feeding that data back to the compiler, to guide further optimization efforts. They have PGO to choose a good parallelization pattern and data chunk size to maximize throughput at the maximum possible latency. Because Sydow's work demands two compilations and some understanding of parallelization patterns, it operates at a lower level than ours, since we wish to create easy-to-use abstractions that are as unintrusive as possible to the developer.

3.2 Distributed

RStream [33], by Fino et al. is a data-processing platform written in Rust. Their results are competitive with custom MPI implementations and have 2 to 20 times higher throughput than Apache Flink (explained in the next Subsection). Our work does not have the goal of creating a full-fledged platform. Rather, we are creating a library that uses Rust's code transformation functionalities.

In [96], Tronge, Pritchard and Brown showed that an implementation of Open MPI's core components used for intra-node communication inf Rust had performance similar to that of the reference implementation in C. In [97], the authors sought to improve the safety of point-to-point communication leveraging the extra guarantees of memory-safe programming languages. To achieve this, they implemented type matching on top of Open MPI, and a UCX-based library written in Rust. These works are lower-level than ours, focusing on programming feasibility, implementation safety and low-level performance. In theory, the distributed part of our work could have been built on top of their work, rather than rsmpi. This is an avenue for possible future works.

Renoir [24], by Martini et al is another data-processing platform written in Rust. It follows the Dataflow pattern we have explained in Section 2.1.2. The authors claim Renoir to be simple and performant, though they expose a large API with many traits that interact with each-other in complex ways. Their traits define the way the nodes communicate and the nature of their processing (map, reduce, etc.). Renoir can be executed both locally and remotely, and we will be using both runtimes in our experiments in Chapter 5. The main difference with our work, besides the fact we also target GPUs, is that we have a

simpler interface, and do not support Dataflow-style programming, focusing instead on simple linear pipelines. Furthermore, as we will see in Chapter 5, `Renoir`'s performance lags behind most other solutions, especially in multi-threaded environments.

3.3 GPUs and Rust

At the time of writing, finding work related to ours regarding the use of GPU in Rust is challenging. Using "*rust AND (gpu OR "graphical processing unit" OR accelerator OR gpgpu*" as a search string in Scopus[1], IEEE Xplore[2], Web Of Science[3], and Dblp[4] yielded a total of 13 publications after filtering out duplicates, non-final publications, letters, and pieces that otherwise have nothing to do with what we are interested in (for example, some physics and engineering publications discussing actual, physical rust). The 13 that were left are all about computer science; however, most of them aren't actually related to our work.

One of the publications is about using computer vision to detect wheat diseases known as "leaf rust", "steam rust", and "strip rust"; another is about a key-value storage engine written in Rust; another merely a comparison between several languages' performance in parallel signal wave propagation simulations, which includes Rust and CUDA; another still about Rust and FPGAs; another about Rust in parallel low-power embedded systems; and finally there is one about binary code patching. This leaves us with a total of 6 publications in the literature. Examining them further, through a quick reading of their contents, we find that most of these are inadequate as well: one uses an LLVM plugin to speed up the computation of derivatives in the GPU; another implemented GPU support for unikernels using an RPC library written in Rust; another just used Rust to process photon data and send it to the GPU, to display experiments in real-time; and another that shows a lattice Boltzmann method solver built with the ArrayFire library, that let the authors compile the solvers into optimized kernels for CUDA, OpenCL, C++, and Rust. The most recent one is the one that introduces Descend [65], a system programming language targeting GPUs that is inspired by Rust's syntax and borrow checking and memory safety ideas. None of them explore the usage of GPUs' compute capabilities *from within* Rust.

Of the 2 works left, [52] is the closest to our work and attempted to implement high-level abstractions to GPU programming in Rust, but it has the severe limitation that was published in 2013, before Rust's first stable release in 2015.

Because of this, in this Section, we've also selected works that are not associated with the scientific literature and exist mainly in the form of libraries (also called `crates`

[1] https://www.scopus.com
[2] https://ieeexplore.ieee.org/Xplore/home.jsp
[3] https://www.webofscience.com
[4] https://dblp.org/

in Rust) and open-source repositories (Subsection 3.3.2). Our goal is to show what has already been produced in this space, even through non-academic means, and how our work can benefit and is different from it.

3.3.1 Related Publications

As mentioned, [52] was published before Rust's first stable release, making it largely obsolete. Nevertheless, we present it here since it is by far the closest to our work. The authors leveraged the new (at the time) LLVM compilation target of PTX, NVIDIA's low-level virtual instruction set for GPUs to extend Rust with support for GPU kernels. The end result bears some similarities with CUDA and OpenCL. For example, much like the `__global__` annotations in CUDA, in [52] kernels must be annotated with `#[kernel]`, as is shown in Listing 3.1.

```
#[kernel]
fn add_float(x: &float, y: &float, z: &mut float) {
    *z = *x + *y
}
```

Listing 3.1: Kernel in Rust example, from [52]

We use annotations in a very similar way in this work (though they are SPar-inspired). Furthermore, they had to manually add a set of intrinsics that map into low-level GPU-specific values, such as the current thread ID, which is commonly used in GPU code. They also briefly mention that certain advanced language constructs will be impossible to reproduce or translate to the GPU and that the compiler will have to include a pass ensuring kernels do not include code containing such constructs. These are both problems we, too, will face. To complete their work, the authors then used these new low-level features to implement common abstractions on top, such as operations on ranges, reductions, and five-point stencil.

Although very promising, [52]'s primary limitation remains the time at which it was written. Rust has since had dramatic changes (in fact, *Listing 3.1 is no longer valid Rust code* since the primitive `float` was renamed to `f32`). Patching the compiler's code generation, as the authors did, is no longer feasible, at least not without a full-time team. Rust's issue tracker at GitHub[5], at the time of writing, routinely reports around 9000 open issues, despite having a large team of full-time contributors, mostly constituted by Rust and compiler writers veterans that are intimately familiar with the code base. Rust has become a very complex language, and maintaining (or adapting, in our case) its compiler is a huge undertaking. This is why our work is based on code transformations; we merely change the written Rust code according to our rules, and then call the regular compiler to create the final binary.

[5] https://github.com/rust-lang/rust/issues

In [15], the authors discuss Rust's theoretical capabilities in GPU programming and compare the Rust's performance to C++'s using the CUDA ecosystem. The paper is ultimately quite limited in its scope and is essentially a comparison between different CUDA runtime implementations. Overall, especially because we do not have access to the codes used in the benchmarks, it is not very useful to us, but it represents *the only instance, in the entire literature* that we were able to find that directly attempts to use a GPU's compute capabilities with Rust.

3.3.2 Non Academic Work

In contrast to the current state of academic publication, Rust enjoys a fairly significant ecosystem for interfacing with the GPU in the form of libraries. At the time of writing, Searching for "gpu" in `crates.io`[6] yields 1190 results, "CUDA" 422, and "OpenCL" 169. Even though many of these libraries may be old, unmaintained, related to GPU graphics rather than compute, or otherwise irrelevant to us, it is still a far more promising result than what we found in the search databases for academic publications. For this Subsection, we've selected the most relevant crates we've found, based on their public presence, measured by the number of total downloads and general repository activity, and their intellectual value (for example, we exclude crates that are merely bindings to an underlying C library, even though they may be important, because there is nothing to be said about them; they are just bindings).

`rust-gpu-tools`

`rust-gpu-tools` [32] was already mentioned in Chapter 2. It is an abstraction layer on top of CUDA and OpenCL and essentially creates a thin wrapper that allows you to set up a computation on both runtimes using a singular macro call. It has the disadvantage that it does not handle the specific differences between the languages for us, so we must take care to write specific CUDA or OpenCL code, as appropriate. On the other hand, this library will greatly simplify the implementation of our proposed abstraction and is an example of how Rust's type system and language features can be used to create convenient GPU programming abstractions.

Rust CUDA

The Rust CUDA Project [88] was what the authors of [15] used to benchmark Rust's CUDA runtime implementation. The project is trying to compile Rust to PTX, in much the same way as [52] attempted to do in 2013 (including the `#[kernel]` annotation). Thus,

[6] https://crates.io/

this project nicely ties our two GPU-related academic works together. Being essentially a continuation of [52], it suffers from the same major problem: it is a very complex, difficult undertaking. Accordingly, development had stagnated: almost 4 years passed without the project seeing any commits, but it appears to have been picked up again recently.

`rust-gpu`

EmbarkStudios are taking a very similar route with `rust-gpu` [28], except their compilation target is SPIR-V [61]. SPIR-V is a binary language made to represent graphical shaders and compute kernel primitives. It is most prominently used with Vulkan [62] in the rendering pipeline, but it can also be used in the compute pipeline, and with OpenCL. It is worth noting that, even if Vulkan is commonly associated with graphical applications, it is entirely possible to use it just for its compute capabilities, though the academic HPC literature hasn't explored this very much, most likely because Vulkan is significantly more complex, verbose, and may have certain missing features from CUDA or OpenCL. Nevertheless, [28] is a more promising attempt than [88], because it has a much larger team of contributors, corporate backing, and commits are still being submitted daily. Our focus is not on creating a new compilation target for Rust, but rather just transforming some Rust code into OpenCL kernels, and thus our scope is entirely different from EmbarkStudios' work. Furthermore, compiling Rust to SPIR-V does not yield an executable binary; the SPIR-V file must then be fed into a Vulkan compute pipeline or OpenCL runtime for it to actually run. This does not simplify or abstract away the difficulties of programming for the GPU, like we wish to do, rather, it is merely creating a new language that can be used for writing programs for Vulkan compute and OpenCL.

3.4 Comparative Table

This Chapter's presentation is summarized in Table 3.1. "MT" stands for "multi-threaded" and "DIST" for "distributed".

Explaining Table 3.1's columns in more detail:

- High-Level – indicates whether we consider the abstraction to be high-level. We have made the case for whether we consider an abstraction high or low level earlier throughout the Chapter.
- API – the API the work exposes to its users. In this context, we call something a "framework" when it is a collection of libraries, or very disparate functionalities that do not make up a single unified API, but several of them. We also use it when the original work refers to itself as such. Note that [15] does not have an API because it

Solution	High-Level	API	MT	DIST	GPU
Rayon [83]	Yes	Parallel Iterators	Yes	No	No
Tokio [95]	Mixed	Framework	Yes	No	No
Ppl [11]	Yes	Library	Yes	No	No
Rust-SSP [80, 81]	Yes	Library	Yes	No	No
RStream [33]	No	Framework	Yes	Yes	No
Renoir [24]	No	Framework	Yes	Yes	No
Sydow et. al. [90]	No	PGO	Yes	No	No
Tronge et. al. [96, 97]	No	Library	Yes	Yes	No
Holk et. al. [52]	No	Compiler Support for PTX	No	No	Yes
Bychkov, A. et. al. [15]	No	None	No	No	Yes
rust-gpu-tools [32]	No	Library	No	No	Yes
Rust CUDA [88]	No	Compiler Support for PTX	No	No	Yes
rust-gpu [28]	No	Library	No	No	Yes
SPar-Rust	**Yes**	**Macro-based DSL**	**Yes**	**Yes**	**Yes**

Table 3.1: Related Work Comparison

is a simple literature review. Note also that [90] does not really have an API. Rather, it presents an approach for doing PGO that others would have to reproduce to attain the same performance gains.

- MT – whether the work offers abstractions for multi-threaded programming.
- DIST – whether the work offers abstractions for distributed programming.
- GPU – whether the work offers abstractions for GPU programming.

To summarize Table 3.1 results, this Master's thesis is the first to provide high-level abstractions for all 3 environments: shared-memory, distributed and GPU. Moreover, it is the first work that provides working, high-level abstractions for general-purpose GPU compute in Rust, and the first one to do it entirely based on macro transformations.

4. A HIGH-LEVEL DSL IN RUST FOR EXPRESSING LINEAR PIPELINES ON MULTI-CORES, CLUSTERS AND GPUS

This Chapter is dedicated to explaining our implementation of SPar's programming model, adapted to the Rust programming language and so-named SPar-Rust. We begin by presenting the adaptations we have had to make to SPar's language to make it work more naturally in Rust. Then we proceed to a formalization of linear pipelines that will guide our implementation in the following Sections. We offer a systematic overview of our approach in Section 4.3, and then present the implementations for every supported parallel architecture. In the final architecture we consider, the GPU, we also explain the special considerations it imposed in our code generation implementation.

This research began with [31] where we first tried to adapt SPar's methodology to Rust. One year later, we created a new version of SPar-Rust [30]. We have already presented these advancements in Section 2.7. Here, we consolidate all previous work, such that programming for all 3 parallel environments can be seamlessly integrated into a single unified API. There are many small, but important differences to what we presented in [30] that were necessary to make this possible. For example, we had to rework the MPI implementation, and introduce a "roundtrip" operation that did not exist in [30], because some applications required creating more than one processing stream, and [30]'s implementation would only allow for a single stream pipeline per application. The multi-threaded version, on the other hand, always allowed for any arbitrary number of stream pipelines. This created a semantic mismatch between the two environments, which we have solved in this work. Finally, a lot of extra work had to be put into us supporting OpenCL code generation for GPUs.

4.1 SPar Language in Rust

Listing 4.1 contains a simple example using most of SPar-Rust's functionalities.

```
1  #[source]
2  fn source(input: Input) -> impl Iterator<Item = SourceOutput> {
3      // sequential code
4      ...
5  }
6  #[stage(State(s))]
7  fn stage1(elem: SourceOutput, s: Stage1State) -> Stage1Output {
8      // sequential code
9      ...
10 }
11 #[stage]
12 fn stage2(elem: Stage1Output) -> Stage2Output {
13     // sequential code
14     ...
```

```
15 }
16 #[sink(Ordered)]
17 fn sink(elem: Stage2Output) -> SinkOutput {
18     // sequential code
19     ...
20 }
21 pub fn process(input: Input) {
22     let replicate_stage_1 = 2;
23     let replicate_stage_2 = 3;
24     let s = Stage1State::new();
25     let stream_iterator = to_stream!([multithreaded:
26         source(input),
27         (stage1(s), replicate_stage_1),
28         (stage2(), replicate_stage_2),
29         sink(),
30     ]);
31     // consume the stream_iterator however you like
32 }
```

Listing 4.1: SPar-Rust simple example

Of the five attributes in the original SPar we presented in Section 2.6.1, only `ToStream` and `Stage` remain. They were renamed to `to_stream` and `stage`, to comply with typical Rust naming conventions. The `Input` and `Output` attributes are no longer necessary because they can be completely inferred by the code. Trivially, `Input` is simply the input to each function, while `Output` is its output. `Replicate` has also been replaced by a simple parameter you can pass to the `to_stream` declarative macro. There are new extra attributes: `source` and `sink`, corresponding to the nodes that will represent the source and sink of the processing pipeline, respectively. Both `stage` and `sink` can accept an optional `State(...)`, which declares that certain input parameters are going to be shared among all instances, as an internal state. This state will be passed as an argument at the `to_stream` call site. The `sink` node also has an optional `Ordered` parameter. This will make the `sink` generated code order its output so that items leave the pipeline in the same order the `source` node sent them in. Finally, in Section 4.7, we will see that `source`, `stage` and `sink` all have optional `OpenCL` and a `Cuda` parameters (with `Map` and `Reduce` as arguments), for generating code transformations that will run on the GPU[1]. This leaves us with the following keywords for SPar-Rust:

- `to_stream` - declarative macro that establishes a stream processing pipeline. The macro's implementation will call each function in the pipeline correctly.
- `source` - attribute macro that transforms the associated function into a linear pipeline source node.
- `stage` - attribute macro that transforms the associated function into a linear pipeline stage node.

[1]We have currently only implemented the transformations for OpenCL, but the attribute already exists

- `sink` - attributes macro that transforms the associated function into a linear pipeline sink node.
- `State(...)` - optional parameter for `stage` and `sink`. Declares that certain parameters are permanent states for that node, set at `to_stream`'s call site.
- `Ordered` - optional parameter for `sink`. Declares that the sink should sort its output.
- `OpenCL` - optional parameter for all attribute macros. Declares this function will execute in an OpenCL environment.
- `Cuda` - optional parameter for all attribute macros. Declares this function will execute in a CUDA environment.
- `Map` and `Reduce` - passed as arguments to `OpenCL` and `Cuda` stages. Currently, these do not do anything. See Section 4.7.

4.1.1 `to_stream` syntax

The special `to_stream` declarative macro connects all nodes in the pipeline. It has the following syntax:

Listing 4.2: `to_stream` syntax

```
to_stream![ENVIRONMENT :
    SOURCE_FUNCTION_NAME (SOURCE_FUNCTION_INPUTS),
    (STAGE1_FUNCTION_NAME (STAGE1_STATEFUL_ARGUMENTS), STAGE1_REPLICATION),
    (STAGE2_FUNCTION_NAME (STAGE2_STATEFUL_ARGUMENTS), STAGE2_REPLICATION),
    ...
    SINK_FUNCTION_NAME (SINK_STATEFUL_ARGUMENTS)
]);
```

Where `ENVIRONMENT` is one of `multithreaded`, `mpi` or `gpu`. All other elements are self-explanatory.

4.2 Linear Pipeline Formalization

We reproduce Figure 2.3 as Figure 4.1 to make it easier to follow.

Let the graph G represent a linear pipeline with V vertices and E edges. Each vertex corresponds to a unit of computation, while each edge represents the communication between them. V contains two special vertices, V_0 and V_f, which stand for the sink

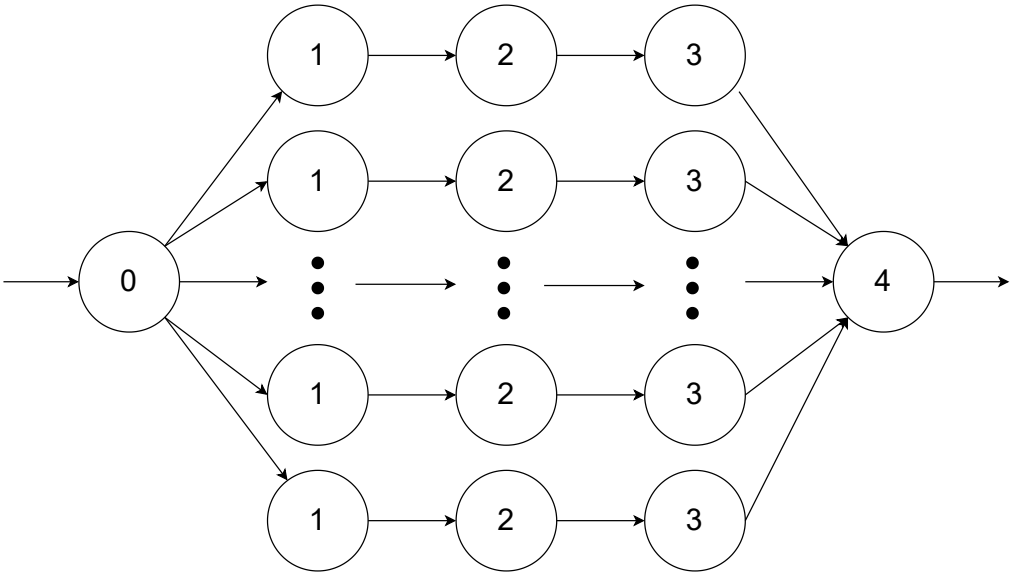


Figure 4.1: A linear pipeline (copied from Figure 2.3)

and the source, respectively (the f in V_f stands for "final"). The operations performed by the sink and the source are unique, meaning they cannot be duplicated in any other vertex in V. Any other computation performed by any other vertex may be duplicated indefinitely. We represent this by annotating the vertices with the same number, as we have done in Figure 4.1. We may also say that vertices with the same number belong to the same *stage* of the pipeline.

Because the final goal is to implement this in a statically typed programming language, let us consider what happens in the pipeline in terms of *datatypes*. Since the edges represent simple communication between independent processes, no edge e can change the underlying type of the data they are transferring. On the other hand, because the vertices stand for arbitrary computation, they *can* change the underlying datatypes. Specifically:

Remark 1 *All vertices labeled with the same number consume the same type of input and produce the same kind of output.*

This puts constraints on how edges and vertices can be connected to build the linear pipeline. All edges that lead to a vertex must be transferring data of the same type. Similarly, for all edges leading *from* a vertex. More formally:

Remark 2 *Let* (v_1, v_2) *represent an edge that starts at* $v1$ *and ends in* $v2$*, and* v_x *is an arbitrary vertex. Given a vertex* $v \in V$*, all edges* (v, v_x) *must transfer the datatype* v_x *expected as input, and all edges* (v_x, v) *must transfer the datatype* v_x *generated as output.*

Remark 3 *Because edges do not transform data, Remark 2 implies that, for every edge* (v_1, v_2)*,* v_1*'s output must be of the same type as* v_2*'s input.*

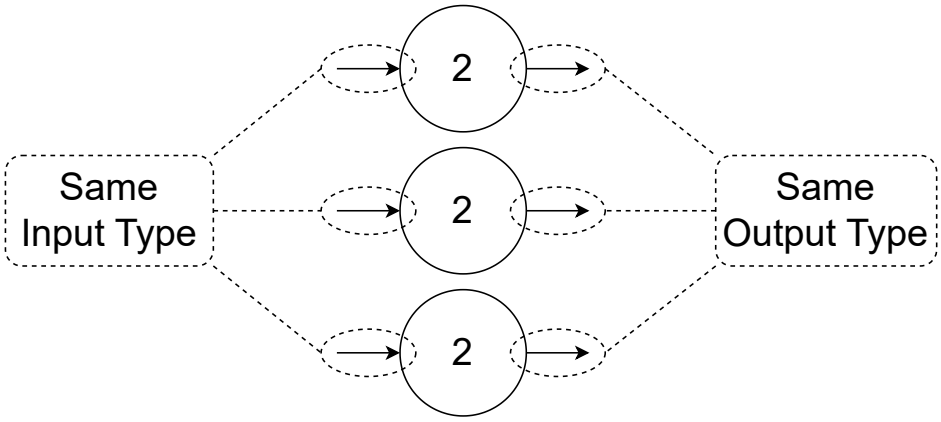


Figure 4.2: Remark 1

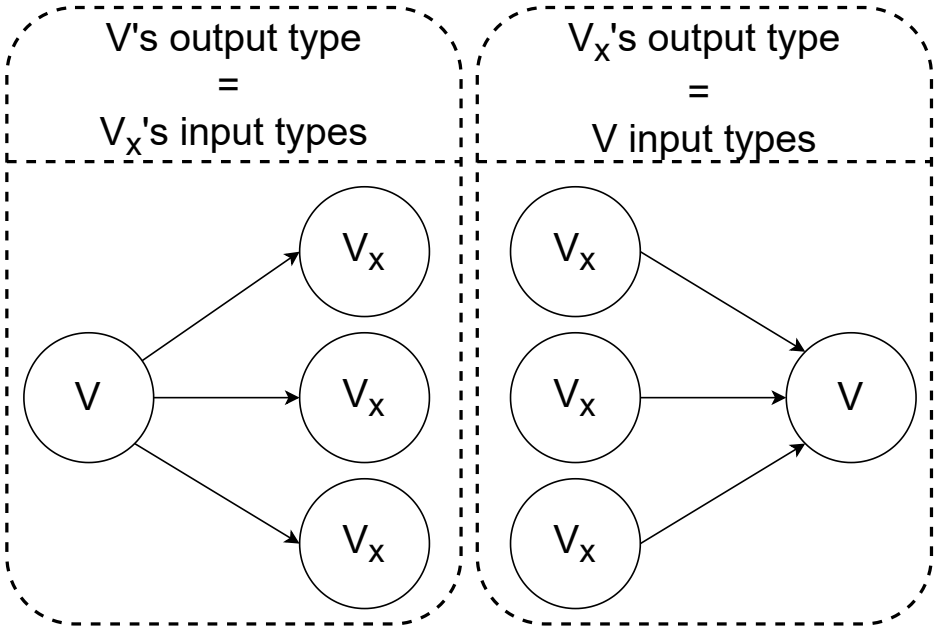


Figure 4.3: Remark 2

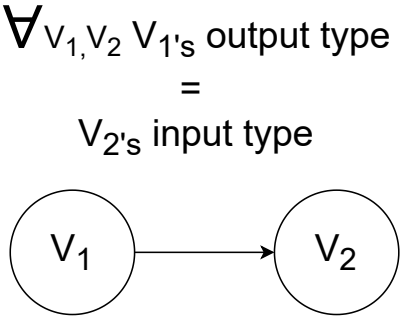


Figure 4.4: Remark 3

Figures 4.2, 4.3 and 4.4 depict these remarks visually.

Given the internal computations of V, Remarks 1- 3 are sufficient to fully specify the types of every element of G. To see why, consider a more straightforward pipeline, without any replication. It would be equivalent to just using the top nodes in Figure 4.1. Once the programmer has specified which computations will occur in each stage, with their inputs and outputs, the type of every edge is forced by Remark 2. If one stage's input does not correspond to its previous stage's output (in other words, if Remark 3 is false), that is a detectable type error made by the developer. To transform the simplified pipeline into the one in Figure 4.1, we can simply create more vertices at any stage between the source and the sink that will execute the exact same code as the other vertices with the same label, and fill in the edges as necessary. Remark 1 guarantees that if we have a valid implementation for one vertex, we can simply replicate it for all vertices in the same stage, and the pipeline will still be valid.

4.3 Systematic Overview

Figure 4.5 shows an overview of the code transformation strategy we will detail in the following Sections.

Note how the multi-threaded and MPI targets will generate the same code in the procedural macro, while the `to_stream` declarative macro will generate different code for every target. Also, note how, when the procedural macro is targeting the GPU, we generate both a rust function and GPU code (see Section 4.7). The GPU code is outputted to a file and standard output, and will be used in subsequent compilations as the code that will execute on the GPU. Within the Rust transformed functions, we will use many traits and structures we will now proceed to define.

4.4 Trait based implementation

This and the following Sections are all dedicated to explaining how the code transformations were implemented. We start by defining general traits that will represent the constraints we have noted in Section 4.2. At the end of this Section, we will show the implementations for these traits for all 3 runtimes.

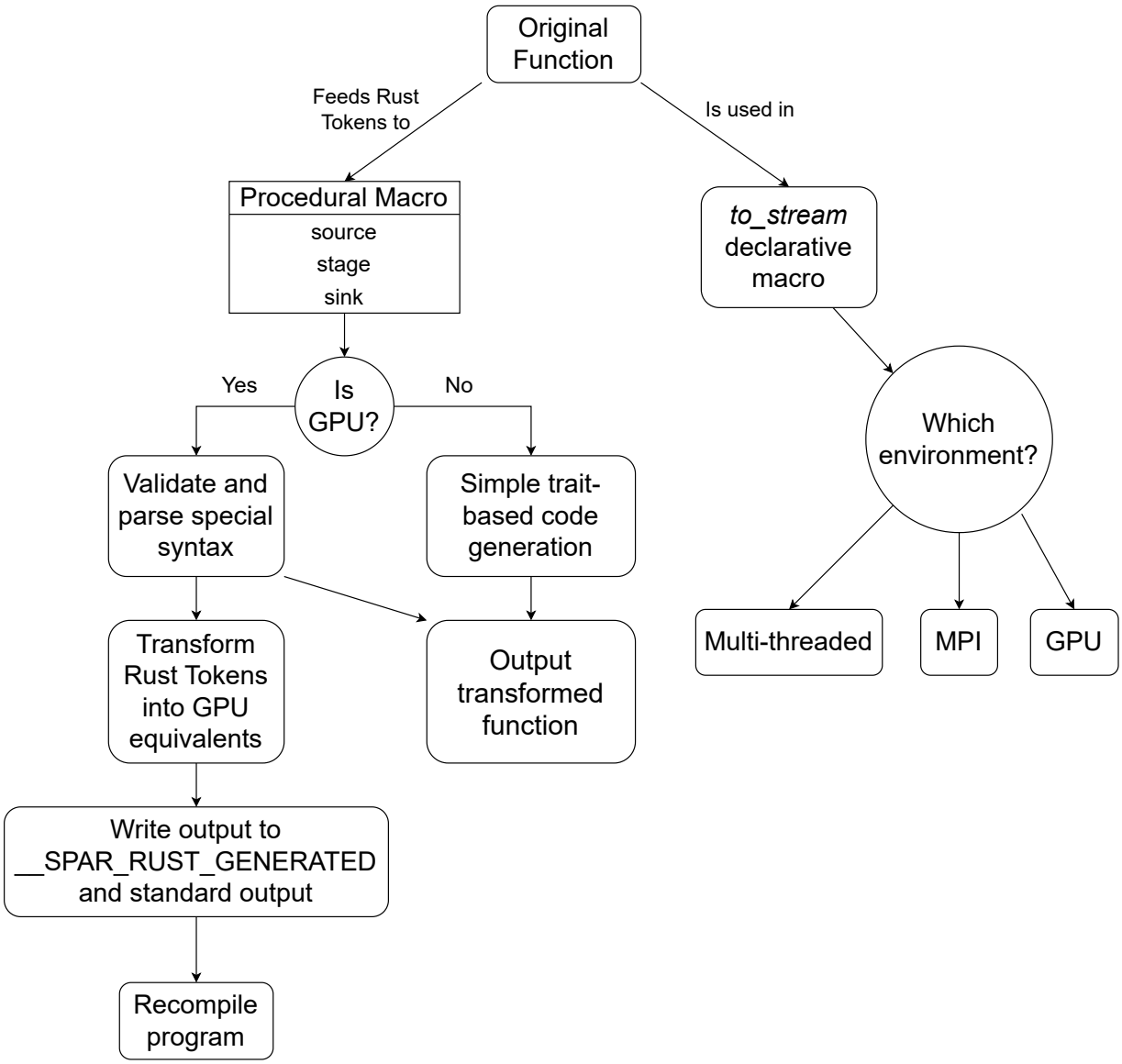


Figure 4.5: SPar-Rust code transformation overview

4.4.1 Edges

The `Sender` and `Receiver` traits (Listing 4.3) represent an edge's ability to send and receive data. Note that the type `T` that `Sender` and `Receiver` communicate, as well as `Sender` and `Receiver` themselves, must all implement the `Send` trait. `Clone` is a trait that indicates a type can be cloned with a `self.clone()` method. The `ReceiveResult` enum lets the implementer define any custom error type, as long as we can print it in a debug context. The `Continue` variant represents a "ping" message, and can be used to implement a "roundtrip" operation, ensuring all messages sent before the roundtrip began have arrived. Now, one needs only define a `struct` that implement these traits for

a given runtime, and the nodes in SPar-Rust's pipelines will be able to communicate with each other in that runtime.

```rust
pub trait Sender<T: Send>: Send + Clone {
    type Error: std::fmt::Debug;

    fn send(&mut self, elem: T) -> SendResult<Self::Error>;
}
pub trait Receiver<T: Send>: Send {
    type Error: std::fmt::Debug;

    fn recv(&mut self) -> ReceiveResult<T, Self::Error>;
    fn into_iter(self) -> ReceiverIterator<T, Self>
    where
        Self: Sized;
}
pub enum ReceiveResult<T, E>
where
    E: std::fmt::Debug,
{
    Ok(T),
    Continue,
    End,
    Error(E),
}
```

Listing 4.3: `Sender` and `Receiver` traits

4.4.2 Workers

Now that we can communicate, we must perform the actual computations. We define, in Listing 4.4, a `Worker` as a data type that has a `Receiver` and a `Sender` of independent types (since, as discussed, nodes can perform arbitrary code transformations).

```rust
pub trait Worker<In: Send, Out: Send>: Sized {
    type R: Receiver<In>;
    type S: Sender<Out> + 'static;

    /// returns this worker's sender
    fn sender(&self) -> &Self::S;

    /// returns this worker's receiver
    fn receiver(&mut self) -> &mut Self::R;

    /// execute the worker's code. Note we accept a function
    /// that is defined elsewhere as an argument.
    fn run<F: FnOnce(Self) + Send + 'static>(self, f: F);

    /// This spawns a compute unit in this worker's
    /// implementation. Note this is only relevant to
    /// the multithreaded implementation, since the
    /// others cannot just fire new units of execution
    /// arbitrarily.
    fn spawn<F>(&self, f: F)
    where
```

```
22          F: FnOnce() + Send + 'static;
23
24      /// Execute the worker's code
25      fn run<F>(self, f: F)
26      where
27          F: FnOnce(Self) + Send + 'static;
28
29      /// This is only relevant to the MPI implementation.
30      /// It allows the workers to ensure all previously
31      /// sent messages have arrived
32      fn roundtrip(&mut self) {}
33  }
```

Listing 4.4: `Worker` trait

Listing 4.4 explains much of its interface in its comments. The existence of both a run and spawn function may be confusing, but will become clearer once we look at the multi-threaded implementation. The above specification also does not include a way of replicating workers. This is because replication of independent execution units is highly dependent on the environment we are developing for. In order to fully understand how replication is achieved for every implementation, we will have to first see the specific implementations of the above traits for every runtime, and combine that with their respective implementation of the `to_stream` macro.

4.4.3 Runtime Implementations

This Subsection shows the implementation of the Sender, Receiver and Worker traits for each of the runtimes we are interested in.

Multi-threaded

```
1  pub struct MtSender<T: Send> {
2      sender: mpsc::Sender<T>
3  }
4  impl<T: Send> Sender<T> for MtSender<T> {
5      type Error = ();
6
7      fn send(&mut self, elem: T) -> SendResult<Self::Error> {
8          match self.sender.send(elem) {
9              Ok(()) => SendResult::Ok,
10             Err(_) => SendResult::End,
11         }
12     }
13 }
14 pub struct MtReceiver<T: Send> { receiver: mpsc::Receiver<T> }
15 impl<T: Send> Receiver<T> for MtReceiver<T> {
16     type Error = ();
17
18     fn recv(&mut self) -> ReceiveResult<T, Self::Error> {
19         match self.receiver.recv() {
```

```
20                Ok(elem) => ReceiveResult::Ok(elem),
21                Err(_) => ReceiveResult::End,
22            }
23        }
24    }
```

Listing 4.5: Multi-threaded `Sender` and `Receiver`

As Listing 4.5 shows, the multi-threaded implementation of `Sender` and `Receiver` is a simple wrapper around the standard libraries `mpsc`[2] channels. Rust's standard library already implement the semantics we need, so our implementation can be very simple. When the standard library's `mpsc` channel element returns an error, it is because the other side of the channel disconnected, so we return a `SendResult::End`, or `ReceiveResult::End`, as appropriate.

```
1  pub struct MtWorker<In: Send, Out: Send> {
2      thread_pool: ThreadPool, // any implementation will do
3      receiver: MtReceiver<In>,
4      sender: MtSender<Out>,
5  }
6  impl<In: Send + 'static, Out: Send + 'static> Worker<In, Out>
7      for MtWorker<In, Out> {
8      type R = MtReceiver<In>;
9      type S = MtSender<Out>;
10
11     /// runs the code in a separate thread
12     fn run<F: FnOnce(Self) + Send + 'static>(self, f: F) {
13         std::thread::spawn(move || f(self));
14     }
15
16     /// spawns a new unit of execution
17     fn spawn<F: FnOnce() + Send + 'static>(&self, f: F) {
18         self.thread_pool.spawn(f);
19     }
20
21     fn sender(&self) -> &Self::S { &self.sender }
22     fn receiver(&mut self) -> &mut Self::R { &mut self.receiver }
23     fn roundtrip(&mut self) { /* NO-OP */ }
24 }
```

Listing 4.6: Multi-threaded `Worker`

Listing 4.6 shows why it is necessary to have both a `run` and a `spawn` method in the `worker` trait. `run` is used to execute the worker's code in a separate thread (otherwise, the pipeline would block while executing just a single worker). `spawn`, on the other hand, is used to actually spawn a unit of execution – a node in the pipeline, doing the actual work. The `ThreadPool` implementation we are using is that of `rayon`'s. The library exposes a low-level interface allowing us to interact directly with its thread-pool. It is, however, possible to use any other implementation that allows us to spawn threads in a pool with a configurable amount of workers. This could be done in future works. We use a thread-pool both because it makes it easier to reuse resources (spawning a thread is a

[2] `mpsc` stands for "multiple producer, single consumer".

costly operation), and because it makes it easier to implement a configurable amount of replication: simply create the thread-pool with the desired amount of threads.

MPI

The MPI implementation is a lot more involved than the multi-threaded one. We will omit logging and simplify other unessential details for didactical purposes.

```rust
use mpi::datatype::Equivalence;
use mpi::point_to_point::Destination;
use mpi::point_to_point::Source;
use mpi::topology::*;

const MPI_TAG_NORMAL: i32 = 0;
const MPI_TAG_ROUNDTRIP: i32 = 1;
const MPI_TAG_END: i32 = 2;

pub struct MPISender<T>
where
    T: serde::Serialize,
{
    /// to whom this Sender sends its messages
    dst: Vec<mpi::Rank>,
    /// the index of dst that will receive the current message
    cur: usize,
}

impl<T> Sender<T> for MPISender<T>
where
    T: serde::Serialize,
{
    type Error = String;

    fn send(&mut self, elem: T) -> SendResult<Self::Error> {
        let i = self.cur % self.dst.len();
        let dst_rank = self.dst[i];

        let world = SimpleCommunicator::world();
        let process = world.process_at_rank(dst_rank);

        let buf = serialize(&elem).unwrap();
        process.send_with_tag(&buf, MPI_TAG_NORMAL);
        SendResult::Ok
    }
}

impl<T> MPISender<T>
where
    T: serde::Serialize,
{
    pub fn roundtrip(&mut self) {
        let world = SimpleCommunicator::world();
        for dst_rank in &self.dst {
            let process = world.process_at_rank(*dst_rank);
            process.send_with_tag(&0u64, MPI_TAG_ROUNDTRIP);
        }
    }

```

```
51      pub fn end(&mut self) {
52          let world = SimpleCommunicator::world();
53          for dst_rank in &self.dst {
54              let process = world.process_at_rank(*dst_rank);
55              process.send_with_tag(&0u64, MPI_TAG_END);
56          }
57      }
58  }
59
60  pub struct MPIReceiver<T>
61  where
62      T: serde::de::DeserializeOwned,
63  {
64      /// keeps track of how many Senders who send to this node
65      /// have not sent an "end" message
66      initial_sources: u32,
67      /// keeps track of how many Senders who send to this node
68      /// have not sent a "roundtrip" message
69      sources: u32,
70      /// buffer to receive messages
71      buf: Vec<u8>,
72  }
73
74  impl<T> Receiver<T> for MPIReceiver<T>
75  where
76      T: serde::de::DeserializeOwned + std::fmt::Debug,
77  {
78      type Error = String;
79
80      fn recv(&mut self) -> ReceiveResult<T, Self::Error> {
81          let world = SimpleCommunicator::world();
82          let mut status = world.any_process().probe();
83
84          while status.tag() == MPI_TAG_ROUNDTRIP {
85              // consume the message
86              _ = world
87                  .process_at_rank(status.source_rank())
88                  .receive_with_tag::<u64>(MPI_TAG_ROUNDTRIP);
89
90              self.sources -= 1;
91              // if we have finished the roundtrip, reset our sources
92              if self.sources == 0 {
93                  self.sources = self.initial_sources;
94                  return ReceiveResult::Continue;
95              }
96              status = world.any_process().probe();
97          }
98
99          while status.tag() == MPI_TAG_END {
100             // consume the message
101             _ = world
102                 .process_at_rank(status.source_rank())
103                 .receive_with_tag::<u64>(MPI_TAG_END);
104
105             self.initial_sources -= 1;
106             // if we have received END from all sources, exit
107             if self.initial_sources == 0 {
108                 std::process::exit(0);
109             }
110             status = world.any_process().probe();
```

```
111         }
112
113         // discover the message's size and make sure our buffer
114         // is big enough
115         let size = status.count(u8::equivalent_datatype());
116         self.buf.resize(size as usize, 0);
117
118         let _status = world
119             .process_at_rank(status.source_rank())
120             .receive_into_with_tag(&mut self.buf[0..], MPI_TAG_NORMAL);
121
122         let de = deserialize(&self.buf).unwrap();
123
124         // if everything went right, return the deserialized element
125         ReceiveResult::Ok(de)
126     }
127 }
```

Listing 4.7: MPI `Sender` and `Receiver`

Listing 4.7 shows the MPI implementation of the Sender and Receiver traits. Note how we demand the type being communicated to always be `Serialize` and/or `Deserialize`, through the "`where T: serde::Serialize`" notation. Our communication strategy is a round-robin, where the sender has a list of MPI ranks it can send to, and will rotate between them as it sends its messages. Experimenting with different ways of distributing tasks (for example, implementing an on-demand scheme) could be done in future work. In [30], we had implemented communication by sending two messages at once: the first containing just a number representing the size of the message, the second the message itself. Using MPI's `probe` functionality, we can detect the message's size by itself, without having to send its length in a previous message. Like this, we halved the amount of communication compared to our previous work.

In this implementation, compared to the multi-threaded one, ending the stream is a more involved operation. The `Senders` will send an `END` message to all the processes they can send to, and each process will exit only after receiving the `END` message from all its sources. This ensures all pending messages will always arrive.

Finally, we have implemented the roundtrip operation (it is a NO-OP in the multi-threaded version). Roundtrip is the same as `END`, but the node won't exit after receiving it. Its purpose is to ensure all previous messages have arrived, without ending the stream. The roundtrip operation did not exist in our previous work [30], which represented a limitation on the kinds of applications it could be used on. Specifically, [30] could only model applications that instantiated a single processing stream, because once the stream processing ended, we would exit the MPI environment automatically. Now, we only begin sending the `END` messages when the application exits[3]. This allows us to create as many processing pipelines as we want, thus achieving parity with the multi-threaded backend.

[3] we use the C `atexit` function to execute code before the program's termination. C's standard library functions can be called in Rust by using the `libc` library (https://github.com/rust-lang/libc).

```rust
pub struct MPIWorker<In, Out>
where
    Out: serde::Serialize,
    In: serde::de::DeserializeOwned,
{
    receiver: MPIReceiver<In>,
    sender: MPISender<Out>,
}

impl<In, Out> Worker<In, Out> for MPIWorker<In, Out>
where
    Out: serde::Serialize,
    In: serde::de::DeserializeOwned + std::fmt::Debug,
{
    type R = MPIReceiver<In>;
    type S = MPISender<Out>;

    fn run<F>(self, f: F)
    where
        F: FnOnce(Self) + Send,
    {
        f(self);
    }

    fn spawn<F>(&self, f: F)
    where
        F: FnOnce() + Send,
    {
        f();
    }

    fn sender(&self) -> &Self::S { &self.sender }
    fn receiver(&mut self) -> &mut Self::R { &mut self.receiver }
    fn roundtrip(&mut self) { self.sender.roundtrip() }
}

impl<In, Out> Drop for MPIWorker<In, Out>
where
    Out: serde::Serialize,
    In: serde::de::DeserializeOwned,
{
    fn drop(&mut self) {
        self.sender.roundtrip();
    }
}
```

Listing 4.8: MPI `Worker`

Listing 4.8 shows two important differences to the multi-threaded implementation. The first is the implementation of `roundtrip`, and the fact that we call it on `drop`. `drop` is called by Rust whenever a variable goes out of scope. So we are ensuring, every time a `MPIWorker` type goes out of scope, we are waiting so that all of the messages it sent have, indeed, arrived.

The second difference is that the `run` and `spawn` implementations are essentially NO-OPs. This is because MPI does not let us easily spawn new processes after the initial

call to `MPI_Init`. So we will have to do replication in a different way. This problem will be solved later, in Section 4.6.1, when we talk about the `to_stream` macro.

GPU

We will discuss the GPU implementation later in Section 4.7. Originally, we had created many NO-OP implementations of the above traits to comply with our defined interfaces. These structures relied on passing simple pointers around, which would be equivalent to just moving a register value in assembly. However, because we are generating code, we have concluded that, as long as the `to_stream` macro exposes the same API for all 3 runtimes, the details of how we achieve parallelism do not matter. As such, there is no reason to incur the extra overhead of wrapping our types in a pointer just for the sake of complying with an interface the user will never interact with. Therefore, we decided to abandon the above trait definitions and work directly with the GPU types themselves, while keeping in mind our considerations in Section 4.2. Note that it *is* possible to define several NO-OP operators for the GPU that represent every trait we defined above, and indeed, that is what we did for our first implementation (this is not, therefore, a problem with our formal framework). We have simply deemed it unnecessary, as it would increase overhead and make some of the code transformations less direct.

A reasonable question that arises from this is whether the same could not be done for the multi-threaded and MPI versions. The main difference is that the multi-threaded and MPI versions actually do work in their implementations; they aren't simple NO-OPs. The reason the GPU implementation would consist entirely of NO-OPs is that we only managed to send data to the GPU within the stream functions (`source`, `stage` and `sink`), and so `Sender` and `Receiver` will never have anything to send or receive. This makes them NO-OPs. Similarly, executing kernels on the GPU will also be done within the stream functions, turning the `Worker` into a NO-OP. **In essence, the main issue with the GPU implementation is that we have moved the `Sender`, `Receiver` and `Worker` logic inside the function we are transforming**. Therefore, we do not need implementations of these traits for the GPU backend.

The implementations had to be moved inside the transformed function because we need more information to make the transformation correctly. Specifically, we need to know whether the type we are receiving is a vector or a scalar. If it is a vector, we must copy it onto a GPU buffer. If it is a scalar, we can send it to the GPU as-is. Detecting this is possible in a procedural macro, but not in a trait implementation, and so implementing the traits is not only unnecessary, but impossible, due to a language limitation. This would be possible if Rust supported specialization, as Rust RFC 1210[4] proposes, but, as of right now, it is not the case.

[4] https://rust-lang.github.io/rfcs/1210-impl-specialization.html

4.4.4 Source and Sink

The `Source` and `Sink` traits are still missing. We will not offer code Listings for them because they are very trivial. `Source` is the same as `Worker`, but without the `receiver`, `spawn` and `roundtrip` functions. `Sink`, on the other hand, has just a method to turn itself into a `Receiver`, so we can call `recv` on it to retrieve the data that is exiting the pipeline.

4.5 Procedural Macro Function Transformation

In this Section, we will present the function transformation implementation *for the multi-threaded and MPI environments only*. The GPU function transformations will be presented later in Section 4.7. This is because the implementation of these procedural macros is for multi-threaded and MPI are identical, while the GPU implementation has many special considerations. After showing the function transformations, we will also touch on ordering considerations, and how stateful computation is implemented.

We call the original function f_o and the generated function f_g. In our transformations, we must not change f_o's internal semantics, lest the results differ from their sequential implementations. This is trivially handled by simply copying f_o's body into f_g[5]. Furthermore, because f_g will use types that implement the traits we discussed in the previous Section, and those have already been adapted to each runtime, the exact same function transformations work for both multi-threaded and MPI. We will now consider what transformations we must do to each function in Listing 4.9, starting with the `stage`.

```
#[source]
fn source(/*inputs*/) -> /*outputs*/ {/* sequential source logic */}

#[stage]
fn stage(/*inputs*/) -> /*outputs*/ {/* sequential stage logic */}

#[sink]
fn sink(/*inputs*/) -> /*outputs*/ {/* sequential sink logic */}

// later, the developer calls to_stream!(), using these functions
```

Listing 4.9: Desired application level code

[5] note this is not possible if the function has to run in the GPU

4.5.1 `stage`

For this function, f_g will accept a *Worker* implementation with a `Receiver` that will receive a tuple consisting of f_o's input and a `Sender` that will send the same type as f_o's output. In between them, we execute f_o to perform the desired computation. Listing 4.10 shows the result of that. As usual, the code is slightly simplified from the actual implementation.

```
// we demand the stage to implement the Worker trait with the
// correct inputs and outputs
fn stage(worker: impl Worker<(/*inputs*/), /*outputs*/>) {
    loop {
        // receive messages in a loop
        match spar_rust_worker.receiver().recv() {
            ReceiveResult::Ok(elem) => {
                let mut sender = spar_rust_worker.sender().clone();
                spar_rust_worker.spawn(move || {
                    let output = {
                        // copy all of f_o's logic here and
                        // execute it with the 'elem' we received
                    };
                    sender.send(output);
                });
            },
            // if we get a continue, do a roundtrip to ensure all
            // previous messages have been received
            ReceiveResult::Continue => spar_rust_worker.roundtrip(),
            ReceiveResult::End => break,
            ReceiveResult::Error(e) => panic!("spar receiver panicked: {e:?}")
      }
    }
}
```

Listing 4.10: `stage`'s transformed function

4.5.2 `source`

The `source` function has an extra requirement to let us transform it: it **must return an iterator as its output**. If it does not, the procedural macro returns an error (this is trivially detectable since we only need to see if the function's signature output begins with `Iterator`). This corresponds to the notion that the source function must create the units of work that will be sent through the pipeline. A function that simply returns an integer does not make sense for the source of a pipeline. Since `source` must return an iterator, f_g just executes f_o and iterates over its results, sending each of those to a worker in the next stage in the pipeline. Listing 4.11 shows what that looks like.

```
fn source(mut src: impl Source<(/*output*/)>) {
    // the original f_o returns an iterator.
    // So we iterate over its results
```

```
4       for item in f_o() {
5           spar_rust_source.sender().send(item);
6       }
7   }
```

Listing 4.11: `source`'s transformed function

4.5.3 `sink`

While the `source`'s f_o function must return an iterator, the `sink`, being the opposite of `source`, generates an f_g that returns an iterator. This allows the developer to iterate over the results outputted by the pipeline. The iterator's implementation will call *receiver*.*recv*() to get the next item, executing f_o before returning it to the caller. Listing 4.12 shows how simple the implementation becomes. The `into_iter` function in line 3 transform the sink into an iterator, by repeatedly calling `snk.receiver().recv()` until it returns `ReceiverResult::End`.

```
1   fn sink(snk: impl Sink</*input*/>) -> impl Iterator<Item = /*output*/> {
2       sink
3           .into_iter()
4           .map(/* call f_o function here to transform every element */)
5   }
```

Listing 4.12: `sink`'s transformed function

4.5.4 Ordering

Some data streams demand the processed items be outputted in the same order as they were inputted. Two typical examples are video frames and compressed data blocks. This means our `sink`'s f_g must allow for that. As mentioned in Section 4.1, to specify that we want the output to be ordered, we use `#[sink(Ordered)]`. When we detect the use of the `Ordered` keyword, we generate code implementing the algorithm described in [44] to order the output. It works by tagging each item in the `source` with a monotonically increasing integer. As a result, every item in the pipeline is actually a tuple `(item, tag)`[6]. Then, in the `sink`, we use a priority queue to determine which item should be outputted next. The current implementation is not robust against data losses throughout the pipeline (it would wait for the missing data forever), though that could be improved in future works.

[6] Because this makes the code harder to read, omitting the tag was one of the simplifications we made in the above code Listings.

4.5.5 Stateful Computations

Some workers may benefit from having some kind of immutable state for performing their computations. For example, they could read a specification from a file. If we do not allow f_g to receive extra variables representing that state, f_o would have to read the specification on every execution, which would clearly be sub-optimal. To accommodate stateful computations, `sink` and `stage` accept an optional `State(args)` argument (`#[stage(State(var1, var2, ...))]`, see Section 4.1). Upon parsing this argument, we make f_g accept extra parameters: one for each element in `args`. The elements in `args` must have the same name as one of the f_o's parameters. They will have the same type in f_g, and be removed from the type list of the stream's element. For example, if f_o had signature "`fn f(p1:Vec<i32>, p2:u32)`", and we declared `p2` as `State(p2)`, f_g's stream element would be just "`Vec<i32>`", and f_g would have an extra "`p2:u32`" parameter, to be set at the call-site.

4.5.6 Serialization

Serializing data for execution in distributed environments is, as mentioned, provied by the `serde` library. As explained in Section2.2.6, serde provides default serialization strategies for all of Rust's fundamental types, and standard library containers that use them (such as `Vec`). Furthermore, using procedural macros, `serde` can recursively define a serialization strategy for any `struct` whose fields are all serializable themselves. So, for example, a `struct` with only vectors of integers could have a serialization implementation derived automatically by `serde`'s procedural macros. However, there are some cases which `serde` can not handle. For example, in the next chapter, for the `face-detector` application, we have had to write the serialization implementation ourselves, since the application is dealing with opaque `structs` that were defined in C code we have to interact with.

4.6 The `to_stream` declarative macro

To create a linear pipeline, one must simply call and concatenate the above trait implementations together. We created a declarative macro called `to_stream` to facilitate this. Its syntax was already explained in Section 4.1.1. Then, `to_stream` is called as depicted in Listing 4.13, generating, for the multi-threaded environment, roughly the code in Listing 4.14.

```
to_stream!(multithread: [
    f_0 (/* stateful args for f_0 */),
    (f_1(/* stateful args for f_1 */), /* number of workers */),
    (f_2(/* stateful args for f_2 */), /* number of workers */),
    ...
    f_i,
]);
```

Listing 4.13: `to_stream` multi-thread example

```
let (snd_0, rcv_1) = create_sender_receiver_pair();
let f_0 = SequentialSource::new(snd_0);
f_0.run(f_0, (/* args for f_0 */));

let (snd_1, rcv_2) = create_sender_receiver_pair();
let worker =
    MultiThreadedWorker::new(/* number of workers */, rcv_1, snd_1);
worker.run(move |w| f_1(w, /* stateful args for f_1 */));

let (snd_2, rcv_3) = create_sender_receiver_pair();
let worker =
    MultiThreadedWorker::new(/* number of workers */, rcv_2, snd_2);
worker.run(move |w| f_2(w, /* stateful args for f_2 */));
...
let sink = SequentialSink::new(rcv_i);
sink // sink is returned at end
```

Listing 4.14: `to_stream` multi-thread generated code

4.6.1 Replication

The `to_stream` macro is responsible for replicating the workers, evident by the fact it accepts the number of workers as a parameter. This is a very different operation for all 3 runtimes.

The GPU runtime simply ignores the parameter. When using the GPU, we execute everything in the same CPU thread, and will use as many GPU threads as the problem will accommodate.

The multi-threaded implementation passes the parameter on to the `Worker` implementation, as we can see in Listing 4.14. The `MultiThreadedWorker::new` function will create a thread-pool with the requested number of threads. The `Worker::spawn` function will then spawn a job in this thread-pool. The thread-pool's size ensures the correct amount of replication.

The MPI implementation is much more involved. In MPI, we establish the number of processes during initialization, and can not trivially change this number during runtime. This means the `to_stream` macro must spawn all necessary processes at once, and coordinate them all so that each process is running the correct function. To this end, we implemented a simple algorithm, as follows:

1. sum all the requested worker numbers. We will call this S.
2. if $S + 1$ is smaller than the number of MPI processes, exit with an error. The +1 is necessary for the source and sink process.
3. any process with rank $\geq S + 1$ exits successfully immediately, since we will not be using it.
4. source and sink will be executed by process of rank 0.
5. let R_i be the requested amount of replication for worker i. Source will send its elements to processes of rank $[1..R_i)$.
6. processes of rank $[1..R_i)$ will send their elements to processes of rank $[R_i..R_i + R_{i+1})$. Processes of rank $[R_i..R_i + R_{i+1})$ will send their elements to processes of rank $[R_i + R_{i+1}..R_i + R_{i+1} + R_{i+2})$. And so on.
7. the processes associated with the final worker all send their elements to process of rank 0.

Remember that the MPI `Sender` and `Receiver` implementations (Listing 4.7) have internal structures that keep track of the processes they send to and receive from. So we can simply initialize every worker with the correct parameters, as explained above, and our implementation will take care of the rest.

4.7 GPU-specific transformations

In this Section we will discuss all the special considerations the GPU implementation imposes on us. In particular, we will present how we do automatic code generation for the GPU, how we prevent extra copies between CPU and GPU buffers, and the several constraints and limitations our implementation imposes on the programmer. Our general strategy was to force the programmer to use certain patterns so that we could parse them predictably when generating the GPU kernels. Much of the following explanations involve revealing a constraint in the way the Rust must be written, followed by what that allows us to do when generating code.

We have mentioned before that the `Map` and `Reduce` keyword currently do not affect code generation. It is important to keep in mind, while reading this Section, that we do not use the `Map` and `Reduce` keywords to generate specialized code for these patterns. Instead, our strategy is to simply transform the Rust code into GPU executable code in the most straightforward way possible. It is the programmer's responsibility to know whether they are currently working with a Map or Reduce pattern, and manually change the code

within f_o accordingly. Therefore, the methodology we will show is technically lower level than that of parallel patterns, since one can use it build the parallel patterns on top of it, making for a relatively generic API, that can be used to program anything, as long as it conforms with our limitations (see Section 4.7.2).

4.7.1 Procedural macros code generation

Generating code that will execute in the GPU is not as straightforward as generating code for multi-threaded and MPI environments. First, we have limitations: we can only transform functions that use fundamental types (`u32`, `i32`, `f32`, etc) or vectors of those fundamental types. Passing fundamental types to the GPU is trivial; we must simply map the Rust type to its GPU equivalent (for example, `u32` $\rightarrow$ `unsigned int`). For the vectors, we can simply initialize a `Buffer` from the `rust-gpu-tools` library with the vector's content.

We demand the first statement in f_o to be in the form "`let mut output = vec![0; <size>]`". We use the expression in `<size>` to discover the output buffer's size. We always generate an output buffer, even if it the function's body does not use it. The second statement in f_o must be in the form "`for global_id in <range>`". We use `<range>` to find how many GPU threads we will have to spawn when creating the kernel. Currently, we assign a different GPU thread to every value in `range`. Making this configurable could be the subject of future work. *The body of the `for` loop will constitute the body of the GPU's kernel*. We then create a kernel with the same name as that of the function being transformed and the required number of threads, and proceed to execute it passing in all the function arguments we transformed.

GPU kernel generated body

The GPU kernel's body is created by copying the Rust code with some syntactic adaptations. Rust code is often similar in syntax to C, which makes this possible. The most important adaptation we make is that "`let`" statements (`let var: i32 = 0`) have their type translated to their GPU equivalent and placed before the variable's identifier (`int var = 0`). This implies they must always include the optional type specification. For a list of all constrains and limitations, see Section 4.7.2.

Besides the syntactic adaptations, we use the `<range>` token in f_o's second statement to discover whether the current thread is out-of-bounds for the kernel. If its global id is either smaller than the range's starting point, or equal to or larger than its ending point, we exit immediately. Otherwise, the global id will serve as an index in the exact same way

it serves as an index in the original Rust code. Listings 4.15 and 4.16 contain a simple example of a vector sum Rust stage function and its generated equivalent in the GPU.

```
#[stage(OpenCL(Map))]
fn stage(a: Vec<u32>, b: Vec<u32>) -> Vec<u32> {
    let mut output = vec![0; a.len()];
    for global_id in 0..a.len() {
        output[global_id] = a[global_id] + b[global_id];
    }
    output
}
```

Listing 4.15: Rust code to be transformed to the GPU

```
__kernel void stage(
    __global unsigned int * a,
    unsigned int a_len,
    __global unsigned int * b,
    unsigned int b_len,
    __global unsigned int * output,
    unsigned int output_len)
{
    unsigned int global_id =
        get_global_id(1) * get_global_size(0) + get_global_id(0);
    if (global_id < 0 || global_id >= a_len ) return;
    output[global_id] = a[global_id] + b[global_id];
}
```

Listing 4.16: GPU generated code

These Listings also show other implementation artifacts: we always transform a Rust vector into a pointer and a length. Also, the `OpenCL` annotation comes with a `Map` argument. One of either `Map` or `Reduce` must be specified at every stage. This is because we originally planned to implement different transformations for the map and reduce processing patterns. Ultimately, these were deemed unnecessary, and the implementation is the same regardless of whether `Map` or `Reduce` is specified. The keywords still exist to facilitate future work where specializing the implementations may be possible. As it currently stands, our implementation is far more amenable to maps than reduces. Writing a reducer would be technically possible: create a chain of several stages, each outputting a vector smaller than the previous one. This would, however, probably be very inefficient. On the other hand, a map can be written naturally, by creating an output vector of the desired size and filling it with the relevant results of your calculations. Note, nevertheless, that it remains the responsibility of the programmer to understand that they are using a map or reduce pattern. We simply transform the Rust code to OpenCL as best we are able to, without taking into account the parallel programming pattern the user is operating with.

The generated Rust function is printed to both standard output and a file called "`__SPAR_RUST_GENERATED_OPENCL`". To perform a full compilation from scratch, it is actually necessary to compile the code twice: the first time will print out the transformation to the generated file, the second will copy the file's content into the binary to instantiate the

GPU kernel. It is not necessary to do a full compilation; one can run just “cargo check” to verify the source’s syntax, as that will already trigger the procedural macro’s execution.

Rust generated code

To send data to the GPU, we must make the appropriate calls to `rust-gpu-tools`. Therefore, the original Rust function must be transformed to do that. Listing 4.17 shows a simplified and commented result of Listing 4.15’s transformation.

```rust
fn stage(
    // every Buffer must be accompanied by its length as a u32
    mut elem: (
        (spar_rust::rust_gpu_tools::opencl::Buffer<u32>, u32),
        (spar_rust::rust_gpu_tools::opencl::Buffer<u32>, u32),
    ),
) -> (spar_rust::rust_gpu_tools::opencl::Buffer<u32>, u32) {
    // rust_gpu_tools demand we execute a closure
    let gpu_execute = |
        program: &spar_rust::rust_gpu_tools::opencl::Program,
        args: (
            (spar_rust::rust_gpu_tools::opencl::Buffer<u32>, u32),
            (spar_rust::rust_gpu_tools::opencl::Buffer<u32>, u32),
        ),
    | -> (spar_rust::rust_gpu_tools::opencl::Buffer<u32>, u32) {
        // We deduce the size of the output buffer based on the output's length
        // in the original code. In this case, the length was the first argument's
        // length, so that is what we use here.
        let output_buffer_len = (args.0.1) as u32;
        // Create the kernel, with as many threads as necessary such that every
        // position in the output buffer is assigned to a different thread.
        let kernel = program
            .create_kernel("stage", ((a_len) as usize).div_ceil(1024), 1024).unwrap();
        // creating the output buffer is unsafe
        let output_buffer = unsafe {
            program.create_buffer::<u32>(output_buffer_len as usize).unwrap()
        };
        // Execute the kernel passing in all arguments.
        // Every buffer is followed by its length.
        kernel
            .arg(&args.0.0)
            .arg(&args.0.1)
            .arg(&args.1.0)
            .arg(&args.1.1)
            .arg(&output_buffer)
            .arg(&(output_buffer_len))
            .run().unwrap()
        (output_buffer, output_buffer_len)
    };
    // this will be the input
    let elem = ((elem.0.0, elem.0.1), (elem.1.0, elem.1.1));
    // create the program with the generated GPU code
    let program = spar_rust::gpu::opencl_program(
        include_str!("../__SPAR_RUST_GENERATED_OPENCL"));
    // run the code with our input
    program.run(gpu_execute, elem).expect("failed to execute worker gpu program")
}
```

Listing 4.17: Rust generated GPU code

Note the transformation is very specific to GPUs, as we do not use any of the traits we defined in the previous Sections. We explained this in Section 4.4.3: the fundamental problem is that we have to generate different code depending on whether the argument we are passing is a vector or a scalar value, but this is currently impossible to detect at the trait implementation layer. And so, we had to resort to *ad-hoc* transformations instead.

Preventing extra copies between the CPU and the GPU

To achieve decent performance, we want to minimize the amount of copying between the CPU and the GPU. To this end, we only copy data from the CPU to the GPU in the source node, and copy it back in the sink node. All intermediate worker nodes receive and send `rust-gpu-tools`'s `Buffers` directly, so data is never copied back to the CPU.

4.7.2 Rust Code Constrains and Limitations

There are several limitations in how the original Rust code may be written so that we can transform it into GPU code using our method. Summarizing all the limitations we have presented in this Section, and adding some missing ones, we arrive at the following list:

1. every input and output must be either a fundamental type, or a vector of a fundamental type;
2. type inference is disallowed. The programmer must always use the "`let i: <type>`" syntax;
3. functions are disallowed. The programmer must manually inline all functions. The exceptions are functions in the standard library's `f32` and `f64` modules, called specifically through the syntax "`f32::name`" and "`f64::name`" (not using imports), whose names match exactly to functions that exist in OpenCL;
4. the first statement in the function **must** be in the format "`let mut output = vec![0; <size>]`";
5. the second statement in the function **must** be in the format "`for global_id in <range>`";
6. after the "`for`" loop's body, there must be only 1 statement: the variables the function will return;
7. no other `for` statements are allowed;

8. `while` and `if` statements' conditions must be surrounded with parenthesis: `while (<condition>)` and `if (<condition>)`. This is not necessary in Rust, but it is in OpenCL and CUDA;

9. no statement can use any type that is not fundamental. The only vector types allowed are the function's input and the the `output` vector; and

10. the only operations allowed on vectors are reading and writing through the indexing syntax: "`vector[<index>]`", and getting the vector's length through "`vector.len()`".

All these constraints ultimately make the Rust code look very non-idiomatic, even if it is still valid Rust code.

4.7.3 `to_stream` implementation

Because the `Sender` and `Receiver` implementations are just NO-OPS, the `Worker` implementation is simple, and we ignore the `replication` argument, the `to_stream` implementation for the GPU runtime is much more straightforward than the others. Call the source function, returning an iterator, and use `map` to call every stage in turn, until we reach the sink (see Listing 4.18).

```
1 source(inputs)
2     .map(|args| stage1(args))
3     .map(|args| stage2(args))
4     /* ... */
5     .map(|args| sink(args))
```

Listing 4.18: `to_stream` generated GPU code

4.7.4 A more complex example

Listings 4.19 and 4.20 show a complex example of generating code for the GPU. It is the Sobel filter of one of our applications in the next Chapter. It shows nearly every limitation we discussed above. We have commented the code (and formatted the GPU code) to make it easier to follow.

We can observe the many limitations imposed on the Rust code: the format of lines 5 and 9 in Listing 4.19 is fixed. The lines *must* always be, first, a mutable vector named "output", followed by a `for` loop. The GPU function is composed primarily of what is inside this `for` loop. Each element of the loop will execute in a different thread. Within the loop, we see that we had to use a `while` loop, even though another `for` would make for more idiomatic Rust code. Furthermore, all the types are specified with the syntax `let`

`var: <type>`, which is also not idiomatic Rust. Finally, we use two functions prefixed by `f64`: `f64::min` and `f64::sqrt`. These functions have direct OpenCL equivalent, and so we can transform them by simply removing their prefixes in the GPU code.

```
fn sobel_filter(img_vec: Vec<u8>, width: u32, height: u32) -> (Vec<u8>, u32, u32) {
    // These first two lines MUST be in this exact general format, as
    // we explained above. We will use this first line to calculate the
    // GPU's output buffer's size
    let mut output = vec![0u8; (width * height) as usize];
    // This line will be used to know how many threads we should spawn.
    // We execute every element in the range in a different thread.
    // In this case, we will be executing 'width' GPU threads.
    for global_id in 0..width as usize {
        let stride: usize = 2 + width as usize;
        // For loops besides the above are not allowed, so we use a while loop instead
        let mut j: usize = 0;
        while (j < height as usize) {
            // Note all the explicit typing with ': i32' and 'as i32'.
            // This is very non-idiomatic Rust, but it is necessary for
            // us to be able to transform it.
            let val0: i32 = img_vec[global_id + (j * stride)] as i32;
            let val1: i32 = img_vec[global_id + 1 + (j * stride)] as i32;
            let val2: i32 = img_vec[global_id + 2 + (j * stride)] as i32;
            let val3: i32 = img_vec[global_id + ((j + 1) * stride)] as i32;
            let val5: i32 = img_vec[global_id + 2 + ((j + 1) * stride)] as i32;
            let val6: i32 = img_vec[global_id + ((j + 2) * stride)] as i32;
            let val7: i32 = img_vec[global_id + 1 + ((j + 2) * stride)] as i32;
            let val8: i32 = img_vec[global_id + 2 + ((j + 2) * stride)] as i32;
            let gx: f64 = ((-val0) + (-2 * val3) + (-val6) + val2 + (2 * val5) + val8) as f64;
            let gy: f64 = ((-val0) + (-2 * val1) + (-val2) + val6 + (2 * val7) + val8) as f64;
            // calling the min and sqrt functions is allowed because
            // they are prefixed by 'f64::'. No other functions can be called.
            let mag: f64 = f64::min(f64::sqrt((gx * gx) + (gy * gy)), 255.0);
            output[global_id + (j * width as usize)] = mag as u8;
            j += 1;
        }
    }
    (output, width, height)
}
```

Listing 4.19: Sobel filter in Rust (commented)

```
// The first four parameters are the same as those of the Rust function.
// The output is a new parameter that all generated GPU functions have.
// Note how every array had a length accompaning it
__kernel void sobel_filter(
    __global unsigned char * img_vec,
    unsigned int img_vec_len,
    unsigned int width,
    unsigned int height,
    __global unsigned char * output,
    unsigned int output_len)
{
    // We begin by calculaing the global Id and seeing if it is in  the Rust's for
    // loop's range. As explained, we will distribute the work by sending each
    // value in the range to a different GPU thread.
    unsigned int global_id = get_global_id(1) * get_global_size(0) + get_global_id(0);
    if (global_id < 0 || global_id >= width) return;
    unsigned long stride = 2 + width;
    // Note how we use a while loop, and not a for loop. This is one of the
```

```
19      // limitations we mentioned above.
20      unsigned long j = 0;
21      while (j < height) {
22          // in here, we use only indexing operations on vectors, which are one of
23          // the only 2 operations allowed on them.
24          int val0 = img_vec[global_id + (j * stride)];
25          int val1 = img_vec[global_id + 1 + (j * stride)];
26          int val2 = img_vec[global_id + 2 + (j * stride)];
27          int val3 = img_vec[global_id + ((j + 1) * stride)];
28          int val5 = img_vec[global_id + 2 + ((j + 1) * stride)];
29          int val6 = img_vec[global_id + ((j + 2) * stride)];
30          int val7 = img_vec[global_id + 1 + ((j + 2) * stride)];
31          int val8 = img_vec[global_id + 2 + ((j + 2) * stride)];
32          double gx = ((-val0) + (-2 * val3) + (-val6) + val2 + (2 * val5) + val8);
33          double gy = ((-val0) + (-2 * val1) + (-val2) + val6 + (2 * val7) + val8);
34          // here, we call the sqrt and min functions. This is allowed because in
35          // Rust these functions are prefixed by 'f64::', which we can transform.
36          double mag = min(sqrt((gx * gx) + (gy * gy)), 255.0);
37          output [global_id +  (j * width)] = mag;
38          j += 1;
39      }
40  }
```

Listing 4.20: Sobel filter GPU generated code (commented and formatted manually)

5. EXPERIMENTS

In this Chapter, we will present several benchmarks to show the effectiveness of our abstraction and implementation. We begin by presenting the applications we will use in our benchmarks. Then, we will briefly discuss the libraries and frameworks we will compare our work to (all of which were already presented in Chapter 3). Then, we will show our results, which include performance and programmability metrics.

5.1 Applications

We have extended and improved *RustStremBench*[1] provided by [81]. The original work contained 4 applications. We removed the mandelbrot benchmark (called `micro-bench`) because we considered it an unorthodox implementation, and too artificial even for a benchmark application. We then added 4 new programs to the benchmark, 2 of which we specifically created to let us benchmark GPU applications. Thus, we have the following list:

1. `bzip2` – performs compression and decompression with the bzip2 algorithm;
2. `face-detector` – uses OpenCV [13] to detect people's eyes from a video;
3. `image-processing` – applies a series of filters to a list of images (uses the `raster` Rust library[2]);
4. `kmeans` – calculates centroids for a list of points, improving results iteratively;
5. `word-count` – counts the number of times a word appears in a stream;
6. `sobel` – applies a sobel filter [59] to a list of images, after turning them into gray-scale. Unlike `image-processing`, here we wrote the filters manually[3], so that we could later adapt them to the GPU easily; and
7. `latbol` – performs a fluid simulation using the Lattice Boltzmann method [17][4]. This is the other application we will execute on the GPU.

These seven applications all have different execution properties, and will give us insight into the libraries' performance. Their execution graphs can be seen in Figure 5.1. For simplicity, the graphs only show parallelism of two workers per stage, but any arbitrary number of workers could be used for any stage.

[1] original source code available at https://github.com/GMAP/RustStreamBench
[2] available at https://github.com/kosinix/raster
[3] the code was adapted from https://github.com/dangreco/edgy/blob/master/src/main.rs
[4] code adapted from https://github.com/ndbaker1/bloe

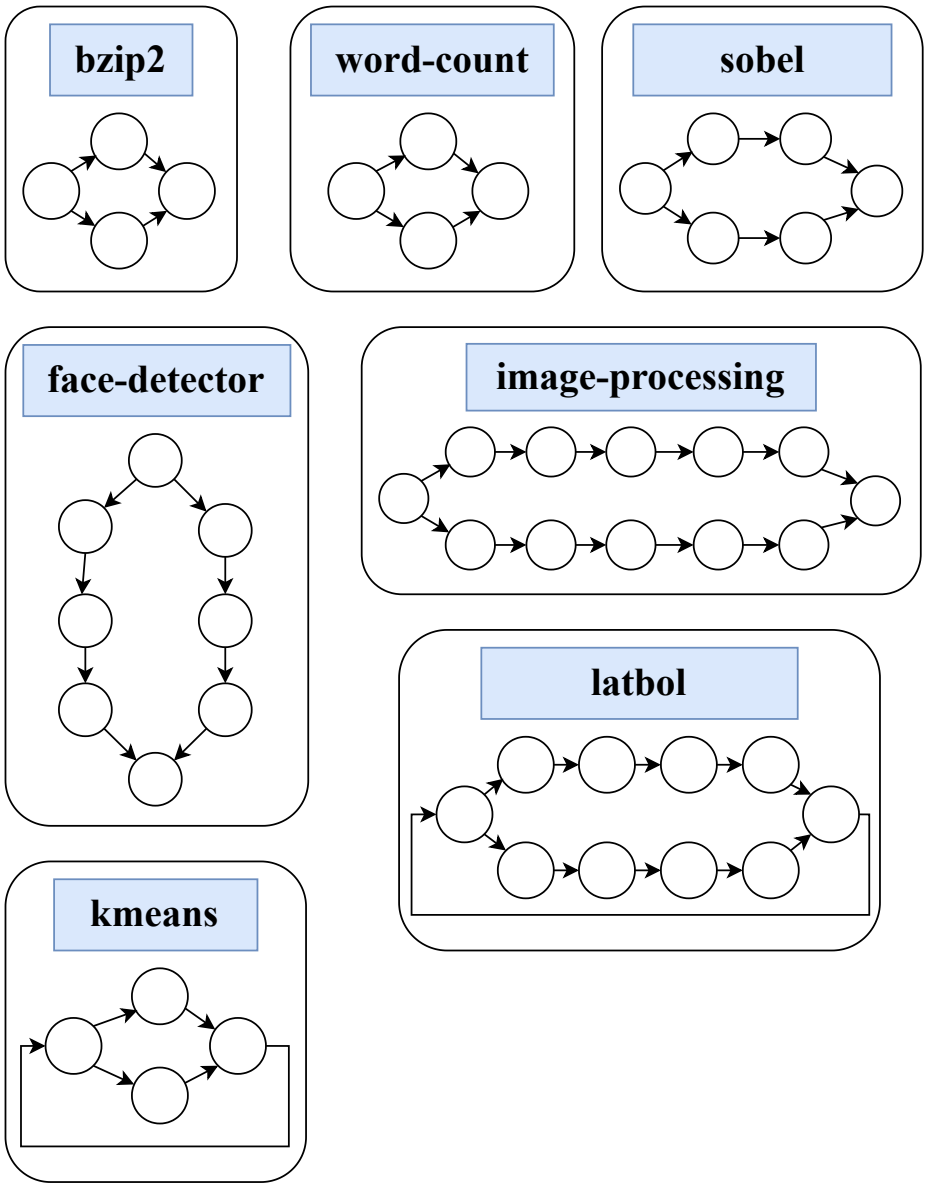


Figure 5.1: Benchmark applications execution graphs.

Figure 5.1 shows that only `bzip2` and `word-count` have the exact same pipeline. Furthermore, `kmeans` and `latbol` are not, strictly speaking, linear pipelines, as we defined them in Section 2.1.1, since they form a cycle. In essence, both these applications run in a loop, with a configurable number of iterations. Each loop iteration is itself a linear pipeline. As we will see in Section 5.3, this has profound implications for the effectiveness of our abstraction.

We will leave discussing the details of the applications' implementations to Section 5.3, where we will mention them as they become relevant to understanding our experimental results.

5.1.1 Verifying correctness

Each application has its own strategy to verify the sequential and parallel implementation generates the same result. For `bzip2` and `face-detector`, we do a simple hash calculation with the SHA256 algorithm. For `word-count`, `image-processing` and `kmeans`, we sort the output after the stream has ended before running the hash algorithm, since the stream itself does not order its elements in these applications. For `sobel` and `latbol`, we also run the SHA256 algorithm, but the GPU applications create slightly different results due to hardware differences in how the CPU and GPU handle floating point operations. Floating point arithmetic is not associative, and so small variations can propagate through the calculations. All GPU implementations' hashes are equal among themselves, but to compare them to the CPU versions, we resorted to printing and manually verifying the numbers. In `sobel`, each pixel value is at most 2 units different from the sequential version, with most pixels being exactly identical, which we consider acceptable. In `latbol`, values are identical up to 4 decimal places, which we also consider acceptable.

5.2 Libraries and Frameworks

For the shared-memory environment, we will use `std-threads`, `rayon`, `tokio`, `ppl`, and `renoir`. We have chosen the first to represent a "manual" implementation using the Rust's standard library facilities; the second and third representing the Rust community default choices for parallelism; the fourth because it is a very recent work of trying to simplify stream-processing in Rust for shared-memory environments; and finally `renoir` was chosen because it is supposed to work in both shared-memory and distributed environments, being therefore the most direct competitor with our own abstraction.

For the distributed environment, we will use `mpi` and `renoir`. As we just mentioned, `renoir` is important because it is trying to do something similar to us. The `mpi` implementation is simply an implementation using `mpi` function calls directly (done with `rsmpi`), without a library to abstract them away. Because `SPar-Rust` also uses MPI in its implementation, we will sometimes use the term "raw MPI" to refer to the pure `mpi` implementation.

Finally, for the GPU, we have just the manually written GPU implementation to compare with `SPar-Rust`. As we explained in Section 3.3, there are not many academic works that try to simplify using GPU compute in Rust, and so there is not much we can use to compare with. Extending the benchmark to use more GPU libraries, when those come to be, can be the subject of future works.

In all cases, we tried to implement the benchmarks in a “normalized” way. That is, all benchmarks use the same parallelization strategy for every library. Any instance where we had to deviate from this will be explicitly mentioned in the following Sections. This also entailed having to rewrite some of the original benchmarks in *RustStreamBench*, and we have used the opportunity to also update their dependencies to their latest releases. The versions we are using for every library is as follows:

- `rayon` – version 1.10;
- `tokio` – version 1.42;
- `ppl` – commit `aeb4589c304d4cb426b5f57261fb530b596635cd`;
- `renoir` – version 0.2[5];
- `rsmpi` – version 0.8; and
- `rust-gpu-tools` – version 0.7.

5.3 Results

For all experiments in this Section, we are using Rust nightly, version 1.86.0, from 2025-01-08, hash a580b5c37. For every environment, we will present, in order, performance and programmability experimental results. Also, for every benchmark, we ensure that all the necessary states are initialized before measuring. This means we read entire files to memory before starting execution. `renoir`, in particular, has facilities to start a processing stream from a file, which could have been a more natural implementation, but we did not use it because the IO could interfere with the measurements.

To measure programmability, we will use two metrics: significant lines of code (SLOC) and Halstead [46] estimated development time. SLOC will be measured with the `scc` tool[6]. Halstead is a method of measuring programming complexity based on tokens: operands and operators. It counts the number of operators and operands, their total occurrences and uses that to calculate a development effort estimate (in estimated hours of work). We will be using Andrade’s work in [6], which extended Halstead to be more suitable for measuring parallel programming complexity. Since Andrade’s tool was focused in C++, we had to add the relevant keywords for Rust and all used libraries manually ourselves.

[5] `renoir`, at the time of writing, just released version 0.4. However, looking at the changes, they should not affect the results in this work, since they affect functionalities we do not use.

[6] available at: https://github.com/boyter/scc

Also, in the programmability sections, we will note which implementations demanded that we write unsafe Rust code. As was explained in Section 2.2, lack of undefined behavior in safe Rust is one of its big selling points. As such, if we ever had to use the `unsafe` keyword in one of our implementations, this should be noted as a failure of the library's API, for not allowing us to write performant code without it.

5.3.1 Multi-Threaded

The multi-threaded experiments were executed on a machine with two Intel(R) Xeon(R) Silver 4210, 144GB of RAM and 4 hard-drives with 2TB in RAID10. The inputs for each application were as follows:

- `bzip2` – A 532MB file of Wikipedia indexes;
- `face-detector` – a 450 frames video showing a crowd with several faces. More faces to detect is more computationally costly;
- `image-processing` – 1000 images, sized 1920 by 1280;
- `kmeans` – 1000000 points, 250 centroids and 100 iterations;
- `word-count` – A file containing 30640350 words;
- `sobel` – the same image set as `image-processing`; and
- `latbol` – a field 10000 by 10000 large, with a circle of radius 10 to block the flow centered at point (50,5000). Executed 100 iterations.

There are a few special implementations in this set of benchmarks:

1. `word-count` with `rayon` is very slow using a `map` followed by a `reduce`, which is the natural programming pattern for this application. Instead, we use a `fold` followed by a `reduce`. In terms of parallel programming patterns, these two correspond to roughly the same thing. But semantically, `fold` will incur in many less copies of the data than `map` in `rayon`'s internal implementation, making it much more efficient.
2. `renoir` does not have a way of sorting the streams' items, nor a stateful sequential sink. This means we can not retrieve the stream elements in sorted order, and can not even implement a sorting algorithm manually, as we have done for the `std-threads` implementations. Instead, for `bzip2` and `face-detector` (the two applications that require sorting), we simply sort the elements after the stream has completed. This will decrease the amount of parallelism used, which would typically make it slower, but it is ultimately a limitation of `renoir`'s current API.

Performance

Figure 5.2 shows the performance results, measured in throughput, of all benchmark programs executed in a shared-memory environment. In all graphs, 0 worker replication stands for the sequential version of the program. Furthermore, every stage in the pipeline is replicated with the same number of threads. Some frameworks do not give us fine-grained control over how many threads will execute in a given stage, since they schedule them dynamically on the fly (like `rayon`). In those cases, we initialized them with $n^o stages \times replication$ threads total. Table 5.1 shows the best throughput we attained with each runtime, alongside the replication factor they used.

Overall, `rayon` emerges as the clear winner in these benchmarks, consistently achieving either the best or close to the best performance. This is expected, as `rayon` has been optimized by the Rust community for running parallel code efficiently in shared memory environments for many years. `tokio` is also very efficient, but its performance does not always match `rayon`, and in `word-count` in particular, greatly lags behind it. Both `rayon` and `tokio` are faster than `std-threads` in almost all cases, because they use optimized scheduling and load-balancing algorithms (such as a thread-pool with work-stealing) to better distribute the computation across the available CPUs.

`renoir` forces our types to implement `Serialize`, as opposed to just Send and Sync, even when running in a multi-threaded environment. This implies it is doing serialization for communication, which can be costly in terms of performance and memory usage. In fact, `renoir` can not execute `image-processing` and `sobel` with the full range of threads because it exhausts the system's memory, and the operating system kills the process. It also can not execute `latbol` at all, exiting with a stack overflow even with just one process. Considering this, it is remarkable how `renoir` achieves performance similar to the best libraries in `kmeans` and `face-detector`. With some optimizations, it is possible that `renoir` could also be competitive in the other benchmarks as well.

`ppl` shows very strange behavior for the `word-count` application. It seems incapable of properly scheduling the work. We tried multiple implementations, all extracted from `ppl`'s word count examples, and this was the best result we could come up with. `ppl` also does poorly on `kmeans` and `bzip2` decompression. In both `word-count` and `bzip2` decompression, the amount of work a single worker has to do for each item received is very light. This suggests that `ppl` is not good at scheduling light workloads. `kmeans`, on the other hand, can be explained by the fact that the application is not a true linear pipeline, which `ppl` is optimized for. Accordingly, it does very poorly.

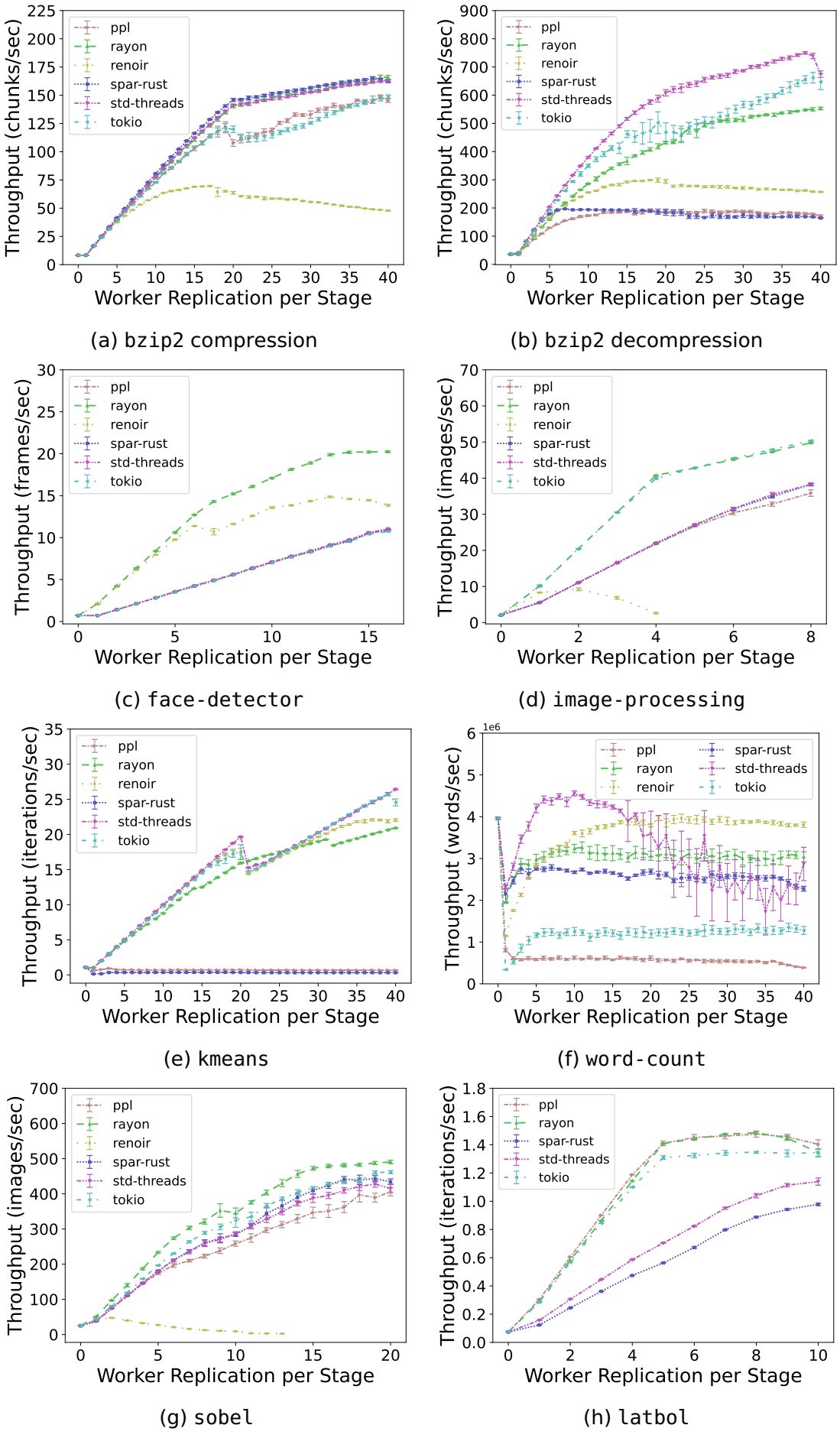


Figure 5.2: Multi-threaded benchmark results

	bzip2 compression		bzip2 decompression		face-detector		image-processing	
	Rep	Thr	Rep	Thr	Rep	Thr	Rep	Thr
std-threads	40	162.75	38	750.07	16	11.04	8	38.28
rayon	39	166.25	40	552.73	16	20.24	8	49.78
tokio	40	148.76	39	661.95	16	10.74	8	50.38
ppl	39	148.58	18	192.28	16	10.93	8	35.87
renoir	17	69.58	18	299.02	13	14.86	2	9.21
spar-rust	38	165.12	7	196.87	16	10.90	8	38.25

	word-count		kmeans		sobel		latbol	
	Rep	Thr	Rep	Thr	Rep	Thr	Rep	Thr
std-threads	10	455.99e4	40	26.41	19	427.86	10	1.14
rayon	11	326.39e4	40	20.94	20	490.55	8	1.49
tokio	38	134.92e4	39	25.71	20	461.65	8	1.35
ppl	1	80.27e4	3	0.92	20	405.87	8	1.48
renoir	24	396.43e4	37	22.08	2	47.56	X	X
spar-rust	7	278.43e4	14	0.36	19	443.18	10	0.98

Table 5.1: Best times for the multi-threaded runtimes.
"Rep" stands for "replication", and "Thr" stands for "throughput".

Finally, `spar-rust`, our own abstraction, does well on most situations. In `bzip2` compression and `sobel`, we match the best performing libraries. In `image-processing`, we match `std-threads` and `ppl`, losing only to `tokio` and `rayon`. In `face-detector`, we match `ppl`, `tokio` and `std-threads`. In `word-count` we are the third slowest library, but we do not display the problematic behaviors of `tokio` and `ppl`. In `bzip2` decompression, we match `ppl` as the worst library, with poor scalability. Once again, decompression has little work per item for every node, implying our abstraction struggles with this kind of workload. In `latbol`, we are the worst library (besides `renoir` which did not even run) because `latbol` has heterogeneous processing stages, with the two outer stages being much more computationally cheap than the inner ones. As such, `latbol` greatly benefits from proper load-balancing, which neither we nor `std-threads` do, thus resulting in us being comparatively slower. At last, `spar-rust`'s performance in `kmeans` is atrocious. This is expected, since `kmeans` is a data-flow application, not a linear pipeline. Therefore, to model it correctly, many concessions had to be made, that ultimately killed its performance. In particular, we re-initialize the library in every iteration, which adds a large constant cost to the computation. We can see that `spar-rust` does get faster as we increase the number of workers up to 20 workers, after which it begins using the virtual cores and its performance flattens. The fact that `spar-rust` is still getting some speed-up up to 20 workers indicates that it can schedule the workload properly, it is just incurring an enormous initialization constant cost at every iteration, making it slower than every other implementation by a large factor.

Table 5.1 must be interpreted alongside Figure 5.2, to evaluate how well each runtime can scale on each application. The results confirm most of Figure 5.2's analysis. It

is interesting that in `bzip2` compression and decompression, `word-count` and `latbol`, the fastest implementations attained their execution speed with a lower replication factor than other implementations, suggesting they could scale even further if we were to increase their workload.

Programmability

Table 5.2 shows the programmability metrics for the multi-threaded implementations. In it, "Hours" stands for Halstead estimated development effort, in hours, and "U" stands for whether the implementation used the `unsafe` keyword in it, with "Y" being "Yes" and "N" "No".

	bzip2			**image-processing**			**face-detector**			**word-count**		
	SLOC	Hours	U	SLOC	Hours	U	SLOC	Hours	U	SLOC	Hours	U
sequential	94	5.15	N	26	0.48	N	50	1.87	N	10	0.11	N
std-threads	171	14.5	N	80	2.75	N	80	9.94	N	35	0.73	N
rayon	106	6.42	N	43	0.99	N	131	7.54	N	22	0.26	N
tokio	126	8.26	N	67	2.09	N	67	4.79	N	30	0.58	**Y**
ppl	121	7.31	N	49	1.27	N	49	4.21	N	14	0.14	N
renoir	173	14.36	N	101	4.56	N	101	11.74	N	29	0.59	**Y**
spar-rust	132	9.75	N	63	1.75	N	63	6.24	**Y**	29	0.92	**Y**

	kmeans			**sobel**			**latbol**		
	SLOC	Hours	U	SLOC	Hours	U	SLOC	Hours	U
sequential	38	1.02	N	40	2.42	N	86	7.59	N
std-threads	60	2.24	N	69	4.38	N	159	15.32	**Y**
rayon	52	1.52	N	49	3.07	N	115	9.73	**Y**
tokio	67	3.16	**Y**	57	3.79	N	128	14.32	**Y**
ppl	39	1.09	N	50	3.46	N	113	8.67	**Y**
renoir	73	2.46	N	60	3.71	N	121	13.38	**Y**
spar-rust	56	2.44	**Y**	57	3.82	N	133	12.74	**Y**

Table 5.2: Multi-threaded programmability metrics.

We see that almost every library is better in terms of programmability than using `std-threads`, with a few exceptions. Because `renoir` demands us to implement the `Serialize` trait for some types, it leads to more code and complexity in `image-processing`, `face-detector` and `kmeans`. `spar-rust` is worse at `word-count` and `kmeans` because the implementations demanded some working around Rust's safety demands, which is why we had to use unsafe in them. The `unsafe` in `face-dectector` for `spar-rust` is merely to implement Sync and Send on some custom types, so it is not a huge problem.

Overall, the two best libraries in terms of programmability seem to be `rayon` and `ppl`. `rayon`'s complexity increases significantly in `face-detector` because we had to interact with its internals to achieve good performance, leading to more verbose code.

Finally, `latbol` demanded we use `unsafe` for every implementation, because we access a vector mutably from multiple threads at once. We calculate the indexes based on the threads' id in such a way that data races will never occur, but Rust's borrow checker can not prove that, and thus it would be impossible to write it in safe Rust. Also, `bzip2` has unsafe code in its implementation, but only to interact with the `bzip2` system library, so we chose to ignore it because it is not a consequence of an API limitation from the libraries we are testing.

5.3.2 Distributed

The distributed experiments were executed on 4 machines connected through 1Gb ethernet, each equipped with two Intel(R) Xeon(R) CPU E5-2620 v3 at 2.40GHz, and 32GB of RAM, for a total of 48 physical cores. The application's inputs remain the same, except for `image-processing`, `sobel` and `latbol`. The first two now use 640 by 427 images and the third a field 3000 by 3000 large. Their inputs had to be reduced because they exhausted the RAM available on the cluster's machines.

In its current format, we could not get `renoir` to work in the cluster. Upon execution of most benchmarks, `renoir`'s stack overflows. This would imply it is storing a lot of data in its stack, or calling many functions recursively until the stack space is exhausted. In any event, we decided to exclude `renoir` from the distributed environment benchmark, as we have found it to be simply too unreliable.

Performance

Figure 5.3 shows the results of executing the experiments in a distributed environment. As in the multi-threaded graphs, the 0 worker replication case stands for the sequential version of the program. Table 5.3 shows the best throughput we attained with each runtime, alongside the replication factor they used.

Figures 5.3a to 5.3d follow the same pattern they did in our previous work[30]. In essence, `bzip2`'s decompression does not need to serialize its messages, since it is already sending raw bytes of data. However, we do not do that optimization in `spar-rust`; we always serialize the data when communicating, which, because the actual work to be performed is relatively small, makes our scalability halt at around a 16 replication factor. `spar-rust` shows very poor performance for `kmeans` for the same reason as in the multi-threaded case: we have to reinitialize the library at every loop, which is even costlier here than it was in the shared memory environment.

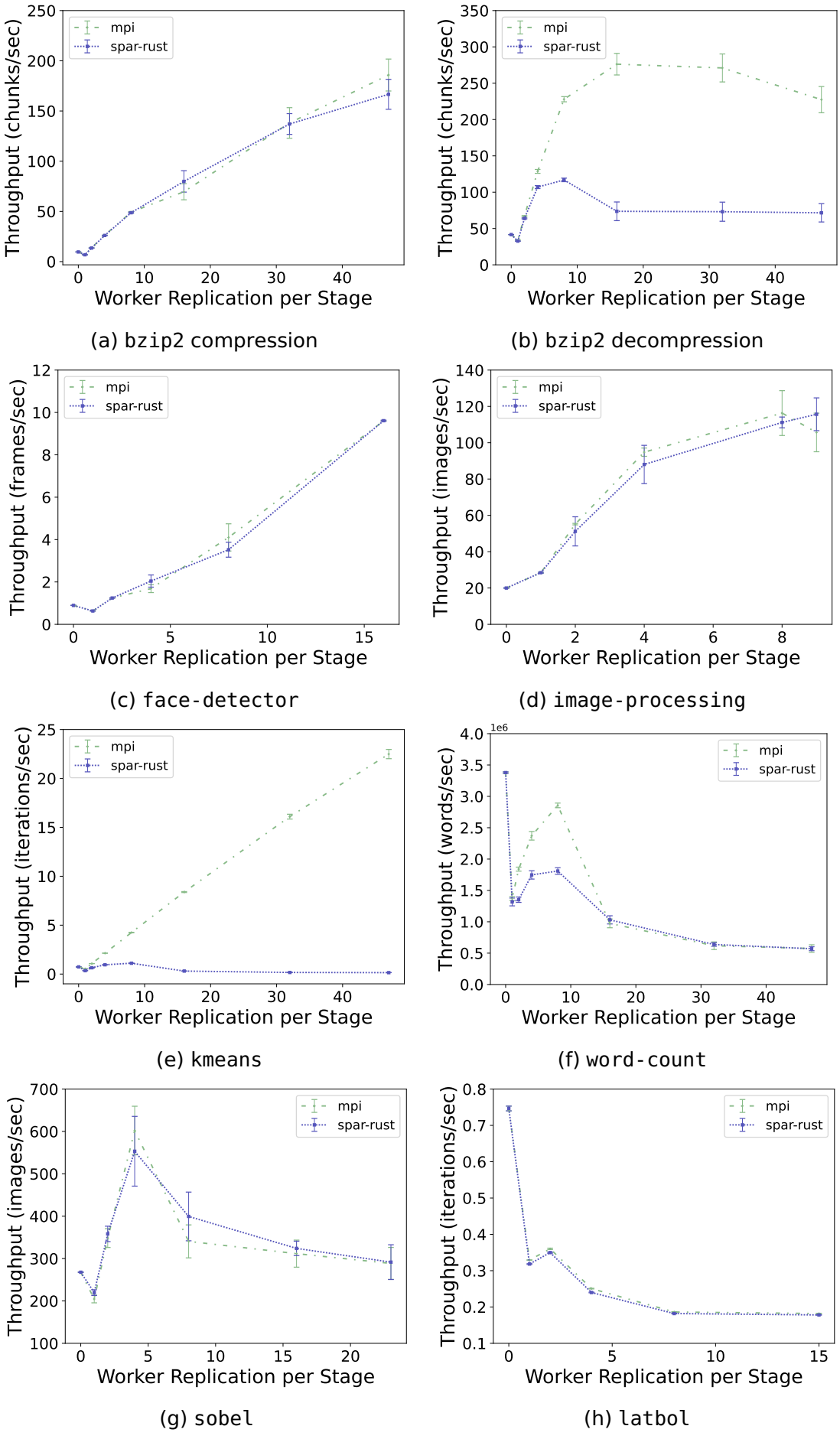


Figure 5.3: Distributed benchmark results

	bzip2 compression		bzip2 decompression		face-detector		image-processing	
	Rep	Thr	Rep	Thr	Rep	Thr	Rep	Thr
mpi	47	185.80	16	276.21	16	9.90	8	116.32
spar-rust	47	166.62	8	116.90	16	9.61	9	115.66

	word-count		kmeans		sobel		latbol	
	Rep	Thr	Rep	Thr	Rep	Thr	Rep	Thr
mpi	8	285.61e4	47	22.50	4	602.16	2	0.36
spar-rust	8	180.96e4	8	1.11	4	553.24	2	0.35

Table 5.3: Best times for the MPI runtimes.
"Rep" stands for "replication", and "Thr" stands for "throughput".

The `sobel` application had limited scalability already in the shared memory environment. Here, its performance peaks at a replication factor of 4. It seems `sobel` can not offer enough work to compensate for the extra communication costs of adding more workers beyond 4 per stage. This is corroborated by the large standard deviations in the graph, indicating the network's instability must have greatly affected our measurements. Nevertheless, `spar-rust` performs equivalently to `mpi` at all points, within error.

`word-count` shows very poor scalability in both implementations. `word-count` was implemented as a pipeline that sends lines of text to every worker. This leads to a lot of small messages being communicated through the system, which causes a large overhead that dwarfs any performance boosts from parallelism. Ideally, a `word-count` application for a distributed environment would instead be programmed by sending much larger chunks of text to each worker. We did not do it this way, because we wanted to maintain symmetry among all implementations. Through the `mpi` implementation could have been made a lot faster, but our goal is not to create the fastest possible `mpi` implementation of the `word-count` application, but rather have an implementation that closely mirrors the strategy `spar-rust` is employing, to measure how much overhead our abstraction is incurring compared to manual implementation.

Finally, `latbol`, like `word-count`, also shows very poor scalability. The explanation for `latbol` is different, however, as, being a fluid simulation, there is a lot of work to be done, as opposed to just counting words. This is further corroborated by how long the sequential execution takes. Rather, as previously mentioned, `latbol` is an application with very asymmetric processing requirements between its stages, with the outer stages being cheap, while the two inner stages are computationally costly. Because we are assigning an equal number of workers for every stage, half the workers are idle most of the time. So we are not fully exploiting the system parallelism. The existence of two cheap stages also implies the communication costs of those stages add enough overhead to compensate for the gains in parallelism. In a production application, we would perhaps merge the two outer stages with the inner stages, thus reducing the amount of idle work-

ers and decreasing communication costs. As mentioned, the application was developed like this to maintain symmetry among all implementations.

These results show that `spar-rust` offers performance on par to that of a manual implementation of the same parallelization strategy, with the exceptions being `kmeans`, since it represents a processing graph `spar-rust` was not designed for, and `bzip2` decompression, because the manual version can skip the data serialization process. This is further corroborated by the results in Table 5.3, where `spar-rust` always matches (or comes reasonably close to) `mpi` for its best timings, except in `bzip2` decompression and `latbol`. Our performance is similar to that of a manual implementation even in degenerate cases, such as in `word-count` and `latbol`, and even shows the same local peaks in performance in cases like the `sobel` application.

Programmability

Table 5.4 shows programmability metrics for distributed environments. We have reproduced the sequential version from Table 5.2 to serve as a reference point.

	bzip2			**image-processing**			**face-detector**			**word-count**		
	SLOC	Hours	U	SLOC	Hours	U	SLOC	Hours	U	SLOC	Hours	U
sequential	94	5.15	N	26	0.48	N	50	1.87	N	10	0.11	N
mpi	255	29.19	N	307	53.89	N	378	58.39	**Y**	93	6.06	N
spar-rust	132	9.75	N	111	4.75	N	205	16.89	**Y**	30	1	**Y**

	kmeans			**sobel**			**latbol**		
	SLOC	Hours	U	SLOC	Hours	U	SLOC	Hours	U
sequential	38	1.02	N	40	2.42	N	86	7.59	N
mpi	102	6.25	N	162	24.15	N	277	55.1	**Y**
spar-rust	69	3.81	**Y**	57	5.33	N	128	15.74	**Y**

Table 5.4: Distributed programmability metrics.

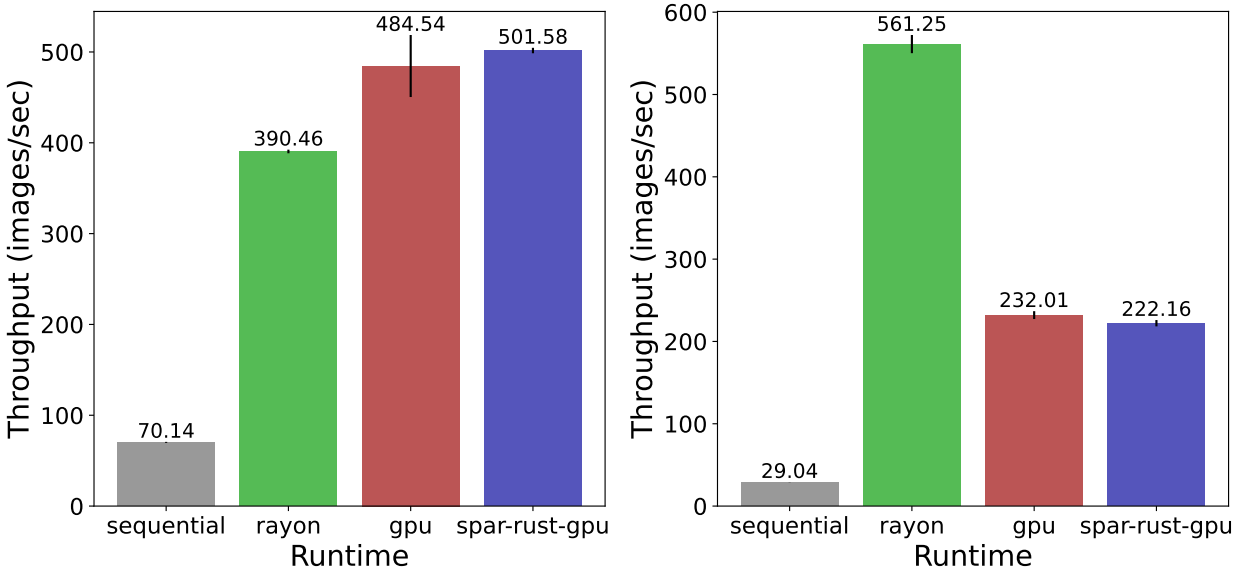
As expected, `spar-rust` is significantly easier to program than `mpi`. The only difference in the `spar-rust` implementations for a distributed system, compared to the shared-memory one, is that we have implement `Serialize` on the types we wish to communicate, and changing the term `multithreaded` to `mpi` in the `to_stream!` declarative macro call. Perhaps more surprising is the fact that `mpi` achieved good performance without using `unsafe` code in cases where it was necessary in `spar-rust`. Unsafe Rust was used in `word-count` to prevent extra line copies that would affect performance, and in `kmeans` to avoid having to duplicate the points and/or the centroids at every iteration. Is it worth noting we are disregarding unsafe calls to `MPI_Finalize` in the `mpi` implementation in nearly all applications. This is because `MPI_Finalize` is only unsafe to call because it is a C function. In actuality, calling it is perfectly fine as long as we exit the

program soon after. Since every `MPI_Finalize` call is immediately followed by a call to `std::process:exit`, we choose to ignore this function when evaluating `unsafe` usage.

5.3.3 GPU

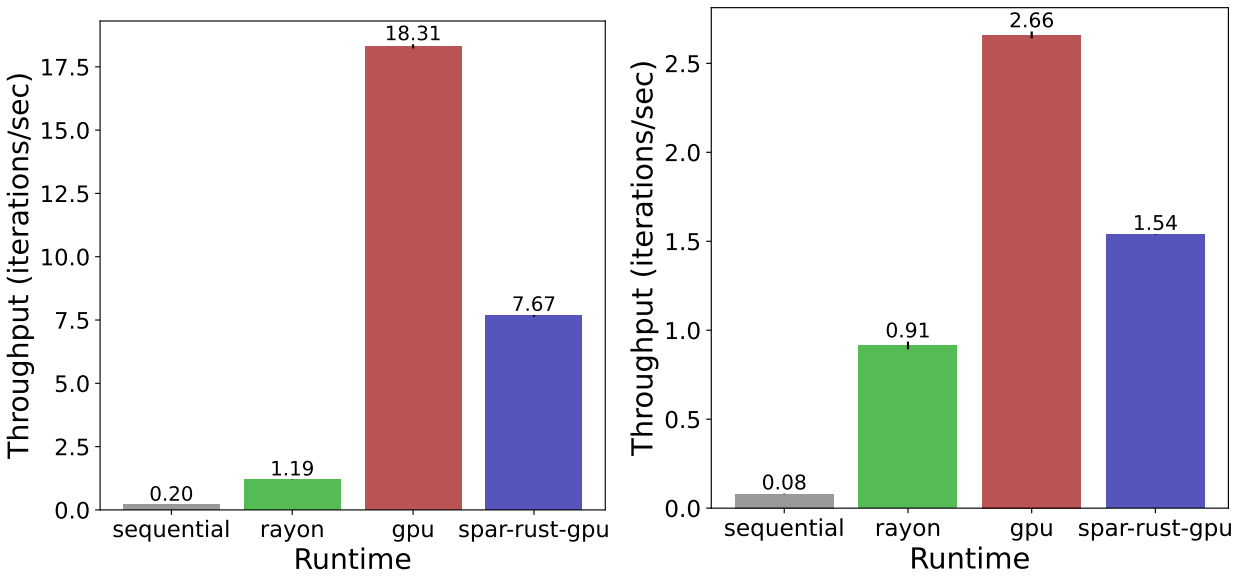
For the GPU, we use just the `sobel` and `latbol` applications, since all others use libraries we can not call from within the GPU. Moreover, we will also include the results of using `rayon` with the maximum number of available threads, to see how well the GPU compares to the most performant CPU implementation. Their inputs are the same as that of the multi-threaded versions. We conducted the experiments in two systems: System A with a AMD Ryzen 5 5600X 6-Core, 32GB of RAM and a NVIDIA GeForce RTX 3090 GPU; and System B with two Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz, 190GB of RAM, and an NVIDIA Tesla M40 GPU. System A, therefore, has a rather weak CPU and a powerful GPU, while System B has two powerful CPUs, and a comparatively weaker GPU, making for two distinct testing environments.

Performance



(a) System A (b) System B

Figure 5.4: `sobel` GPU benchmark application results



(a) System A

(b) System B

Figure 5.5: `latbol` GPU benchmark application results

Figures 5.4 and 5.5 show the results of executing the programs on both machines. We call gpu the manual GPU implementation in OpenCL we use to compare `spar-rust` to. Figure 5.4 shows that in System B, which contains two powerful CPUs and a relatively weaker GPU, we can actually attain greater throughput by executing the program in the CPUs with the maximum number of threads instead. On the other hand, System A already benefits from executing on GPU, being slightly faster than `rayon`. `spar-rust` performs very closely to the manual implementation. The `latbol` benchmark is where we can see more clearly the advantages os the GPU. In System A, GPU timings are 20 times more efficient on average than using `rayon` with the maximum number of threads. `spar-rust` is around 2.5 times less efficient than the manual GPU implementations. In System B, the GPU is still around 3 times faster than `rayon`, with `spar-rust` being around 1.75 slower than the manual implementation.

Programmability

When comparing programmability for the GPU environment, it is important to keep in mind that OpenCL demands we program in two languages: Rust and OpenCL (which is a modified C). Furthermore, `spar-rust` demands its code be written in a very specific way (see Section 4.7.2). These factors make measuring programmability effort very unreliable, since they do not take into account the extra cognitive load of having two languages in the same project, or the extra work that writing code conforming to `spar-rust`'s limitations entails. Having made these caveats, the results can be seen in Table 5.5.

Halstead estimated development hours give an edge to `spar-rust`. Note we had to use `unsafe` to make `latbol` fast on `spar-rust`. Furthermore, we actually did not use

	sobel			latbol		
	SLOC	Hours	U	SLOC	Hours	U
sequential	40	2.42	N	86	7.59	N
gpu	95	9.06	**Y**	199	75.8	N
spar-rust	58	8.1	N	311	55.44	**Y**

Table 5.5: GPU programmability metrics.

the `to_stream` macro in the `latbol` implementation, as that would cause the GPU buffers to be transferred over at every iteration. Instead, we called the transformed functions directly, something only a very advanced user of SPar-Rust would do. Also, because the Rust code that will be transformed into GPU code can not make arbitrary function calls, it leads to a lot of code duplication that could be avoided in OpenCL, which explains the extra SLOC in `latbol`. Despite all these extra complications, Halstead still estimates it would be easier to program with SPar-Rust than with the raw GPU bindings.

SPar-Rust's main strength lies in the fact that it works well by both performance and programmability metrics on all three different environments. Between multi-threaded and distributed, in particular, only minimal changes to the source code must be made. Even though SPar-Rust may not be the most performant or more straightforward to program in all cases, it remains the only programming abstraction that allows such flexibility.

6. CONCLUSION

In this work, we have presented SPar-Rust, a new high-level DSL in Rust for expressing linear pipelines on multi-cores, clusters, and GPUs. We presented a simple formalization of linear pipelines that guided our DSL implementation, and we have shown the runtime and code generation implementation for all target architectures, where choices and limitations were discussed. We also conducted experiments demonstrating our abstraction behaves efficiently in most scenarios. Our most significant limitation remains the one represented by the `kmeans` application in the previous Section, that is, applications that can not be entirely modeled as a linear-pipeline.

Rust's procedural macros proved capable of performing all code transformations we needed in this work. In the original SPar, the authors were forced to develop a whole, albeit simplified, C++ compiler to make their code transformations. Rust gives the programmer enough power with its macro system, and this was not necessary here. However, using procedural macros is still not trivial. The input is raw Rust tokens, meaning it is up to the developer to parse them. More importantly, the *output* must be valid Rust code, which can be complicated due to the many rules it imposes on the programmer. Generally speaking, procedural macro transformations should try to limit themselves to simple, repeatable routines, transforming code in predictable ways, with well-defined rules. Looking at Section 4.5, the procedural macros we used in this work for both the shared-memory and distributed environments generate the same code. The code itself is simple, because it is based on our formalization, which is also very simple.

Generating code for the GPU was very challenging because of the syntactic and semantic differences between the two environments. In this work, we only managed to do it after severely limiting how the Rust code could be written. Overall, this would seem to indicate that GPU abstractions in this direction may not be feasible in ways that are both efficient and not restricting. The GPU has many different properties from the CPU, and so it makes intuitive sense that approaches that work well in one environment may not work as well in the other. Section 5.3.3 shows our abstraction loses around 2 times performance, even with all the concessions we have had to make for our approach to be even feasible. Efficient GPU programming abstraction must take into consideration all of the GPU's idiosyncrasies and find a way to explore them well. A simple abstract formalization is not enough to achieve that.

6.1 Limitations

Our work has many limitations, most of which we have presented throughout this document. Section 4.7.2 lists the several constraints that GPU code generation imposes

on how the function to be transformed can be written. Furthermore, as mentioned, we do poorly on applications that can not be fully modeled with a linear pipeline. We also do poorly in the distributed environment, when a manual implementation can bypass the data serialization step, as is the case in the `bzip2` application. 3 of our 7 applications in Chapter 5 show poor scalability in a distributed environment. Ideally, we would like to have more programs that can scale reasonably in a cluster so that we can conduct more thorough testing for the distributed runtime. In the same vein, ideally, we would also like to include more GPU applications with different levels of complexity.

When it comes to load distribution, we only implemented a round-robin strategy for the distributed environment. Also, our shared-memory implementation does not fully exploit work-stealing on all stages since each stage has its own thread pool.

Finally, our programmability metrics are somewhat fragile. Ideally, we would like to measure this by having a group of people try to create parallel programs with all the mentioned frameworks. Then, we would collect metrics such as: development time, subjective perceived difficulty by the participants, correctness and amount of bugs, and so on. However, finding the resources to conduct this kind of experiment can be challenging.

6.2 Future Work

Throughout this Thesis, we have made many mentions of possible future work. Many of these are small and incremental improvements over the work we have presented here. Nevertheless, there are a few major directions one could take with this research:

- first, one could experiment with relaxing the constraints on code that we can transform to execute on the GPU. This would be very challenging as each constraint we added in our implementation allowed us to make extra assumptions when generating the GPU code. In practice, we would have to develop a static analyzer to understand what the sequential code is doing and try to reproduce that in the GPU. Some constraints will be easier to relax than others. For example, the restriction that the function may contain only one `for` loop could be relaxed by simply running the same parsing logic twice, with the correct adaptions. On the other hand, it is likely that other restrictions will not be able to be relaxed at all This would represent a great advancement in automatic GPU code generation, and would allow a more seamless integration of the three parallel runtime implementations.

- a second, different, direction, would be to explore options to do automatic load-balancing for our implementations. Many of our execution problems could be solved by having more efficient load-balancing techniques. This would possibly involve developing a more complex formal framework, to account for possible imbalances in

the execution graph. We would also have to adapt the implementations with new ideas that supported load-balancing methods. A similar, but ultimately different, idea is to try deducing replication levels for every stage automatically, based on their internal computations. This could also improve performance by better distributing computational resources.

- a third possibility would be to explore leaning more into the parallel patterns paradigm. We could generate specific, optimized code for map and reduce patterns for all implementations. The keywords and syntax to support this in SPar have already been worked out, as we showed in Section 4.1. As in the previous item, this would also entail revisiting our formal framework, and adapting it to work with all these specific patterns. We would then have to implement these patterns for all parallel runtimes.
- finally, we could try to extend SPar's syntax so that it could handle Data Flow processing patterns, as opposed to just Linear Pipelines. This would also contribute to the original SPar's implementation, which also struggles to handle data flow patterns. This would be a major improvement on the language's expressivity and reach in terms of applications that could be modeled with it.

6.3 List of Published Papers

The following list contains the papers we have published during our Master's program, in chronological order. The most important ones are 1 and 4, which correspond to the advancements in SPar-Rust's implementation:

1. **Source-to-Source Code Transformation on Rust for High-Level Stream Parallelism** [31] – accepted and presented at the 27th Brazilian Symposium on Programming Languages (SBLP), in 2023. This paper presents the first version of SPar-Rust (corresponding to Listing 2.9 in Section 2.7). It only implements a multi-threaded runtime.
2. **Analyzing C++ Stream Parallelism in Shared-Memory when Porting to Flink and Storm** [49] – presented at a workshop in the 2023 International Symposium on Computer Architecture and High Performance Computing (SBAC-PADW). This paper reports our findings when comparing SPar applications to their equivalents in Flink and Storm, concluding that SPar is more performant, and makes better use of the machine's resources.
3. **An internal domain-specific language for expressing linear pipelines: a proof-of-concept with MPI in Rust** [30] – accepted and presented at the 28th Brazilian Symposium on Programming Languages (SBLP), in 2024. This paper presents

the second version of SPar-Rust (corresponding to Listing 2.10). This represented an advancement compared to our previous work in 2023, since we managed to create a much more rigorous implementation method, which we used to implement both a multi-threaded and distributed (MPI) runtime.

4. **Automatic Synthesis of Specialized Hash Functions** [48] – accepted at the International Symposium on Code Generation and Optimization (CGO), in 2025. This was a joint project with the Federal University of Minas Gerais and Professor Fernando Magno Quintão Pereira.

REFERENCES

[1] Akidau, T.; Chernyak, S.; Lax, R. “Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing”. O’Reilly Media, Inc., 2018, 1st ed., 352p.

[2] Aldinucci, M.; Danelutto, M.; Kilpatrick, P.; Torquati, M. “Fastflow: High-Level and Efficient Streaming on Multicore”. John Wiley & Sons, Ltd, 2017, chap. 13, pp. 261–280.

[3] Amdahl, G. M. “Validity of the single processor approach to achieving large scale computing capabilities”. In: Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, 1967, pp. 483–485.

[4] Andrade, G.; Griebler, D.; Santos, R.; Danelutto, M.; Fernandes, L. G. “Assessing Coding Metrics for Parallel Programming of Stream Processing Programs on Multi-cores”. In: 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2021), 2021, pp. 291–295.

[5] Andrade, G.; Griebler, D.; Santos, R.; Fernandes, L. G. “A parallel programming assessment for stream processing applications on multi-core systems”, *Computer Standards & Interfaces*, vol. 84, March 2023, pp. 103691.

[6] Andrade, G.; Griebler, D.; Santos, R.; Kessler, C.; Ernstsson, A.; Fernandes, L. G. “Analyzing Programming Effort Model Accuracy of High-Level Parallel Programs for Stream Processing”. In: 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2022), 2022, pp. 229–232.

[7] Andrade, H. C. M.; Gedik, B.; Turaga, D. S. “Fundamentals of Stream Processing: Application Design, Systems, and Analytics”. Cambridge: Cambridge University Press, 2014, 529p.

[8] Andrews, G. R. “Foundations of Parallel and Distributed Programming”. USA: Addison-Wesley Longman Publishing Co., Inc., 1999, 1st ed., 664p.

[9] Arnold, K.; Gosling, J. “The Java programming language (2nd ed.)”. USA: ACM Press/Addison-Wesley Publishing Co., 1998, 442p.

[10] Bataille, M. “Something old: the Gamma 60 the computer that was ahead of its time”, *SIGARCH Comput. Archit. News*, vol. 1–2, Apr. 1972, pp. 10–15.

[11] Besozzi, V. “PPL: Structured Parallel Programming Meets Rust”. In: 2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), 2024, pp. 78–87.

[12] Blessing, J.; Specter, M. A.; Weitzner, D. J. “You Really Shouldn’t Roll Your Own Crypto: An Empirical Study of Vulnerabilities in Cryptographic Libraries”, *arXiv*, 2021, pp. 15, 2107.04940.

[13] Bradski, G. “The OpenCV Library”, *Dr. Dobb’s Journal of Software Tools*, vol. 25, 2000, pp. 120, 122–125.

[14] Bychkov, A.; Nikolskiy, V. “Rust Language for Supercomputing Applications”. In: Supercomputing, Voevodin, V.; Sobolev, S. (Editors), 2021, pp. 391–403.

[15] Bychkov, A.; Nikolskiy, V. “Rust Language for GPU Programming”. In: Supercomputing, Voevodin, V.; Sobolev, S.; Yakobovskiy, M.; Shagaliev, R. (Editors), 2022, pp. 522–532.

[16] Bytecode Alliance. “Cranelift”. Source: https://cranelift.dev/, January 2025.

[17] Chen, S.; Doolen, G. D. “LATTICE BOLTZMANN METHOD FOR FLUID FLOWS”, *Annual Review of Fluid Mechanics*, vol. 30–Volume 30, 1998, 1998, pp. 329–364.

[18] Christophides, V.; Efthymiou, V.; Palpanas, T.; Papadakis, G.; Stefanidis, K. “An overview of end-to-end entity resolution for big data”, *ACM Computing Surveys (CSUR)*, vol. 53–6, 2020, pp. 1–42.

[19] Cook, K. “Git Pull that introduces Rust to the Linux Kernel”. Source: https://lore.kernel.org/lkml/202210010816.1317F2C@keescook/, Oct 2022.

[20] Culler, D.; Singh, J. P.; Gupta, A. “Parallel Computer Architecture: A Hardware/Software Approach”. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, 1056p.

[21] Czech, Z. J. “Introduction to Parallel Computing”. Cambridge University Press, 2017, 354p.

[22] Dagum, L.; Menon, R. “OpenMP: An Industry-Standard API for Shared-Memory Programming”, *IEEE Comput. Sci. Eng.*, vol. 5–1, Jan. 1998, pp. 46–55.

[23] Darlington, J.; Field, A. J.; Harrison, P. G.; Kelly, P. H. J.; Sharp, D. W. N.; Wu, Q.; While, R. L. “Parallel programming using skeleton functions”. In: PARLE ’93 Parallel Architectures and Languages Europe, Bode, A.; Reeve, M.; Wolf, G. (Editors), 1993, pp. 146–160.

[24] De Martini, L.; Margara, A.; Cugola, G.; Donadoni, M.; Morassutto, E. “The Renoir Dataflow Platform: Efficient Data Processing without Complexity”, *Future Generation Computer Systems*, vol. 160, 2024, pp. 472–488.

[25] del Rio Astorga, D.; Dolz, M. F.; Fernández, J.; García, J. D. “A generic parallel pattern interface for stream and data processing”, *Concurrency and Computation: Practice and Experience*, vol. 29–24, 2017.

[26] Donovan, A. A.; Kernighan, B. W. “The Go programming language”. Addison-Wesley Professional, 2015, 400p.

[27] Donovan, A. A.; Kernighan, B. W. “The Go Programming Language”. Addison-Wesley Professional, 2015, 1st ed., 400p.

[28] EmbarkStudios. “rust-gpu”. Source: https://github.com/EmbarkStudios/rust-gpu, January 2025.

[29] Enmyren, J.; Kessler, C. W. “SkePU: a multi-backend skeleton programming library for multi-GPU systems”. In: Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications, 2010, pp. 5–14.

[30] Faé, L.; Griebler, D. “An internal domain-specific language for expressing linear pipelines: a proof-of-concept with MPI in Rust”. In: Anais do XXVIII Simpósio Brasileiro de Linguagens de Programação, 2024, pp. 81–90.

[31] Faé, L.; Hoffmann, R. B.; Griebler, D. “Source-to-Source Code Transformation on Rust for High-Level Stream Parallelism”. In: XXVII Brazilian Symposium on Programming Languages (SBLP), 2023, pp. 41–49.

[32] Filecoin Project. “rust-gpu-tools”. Source: https://github.com/filecoin-project/rust-gpu-tools, January 2025.

[33] Fino, A.; Margara, A.; Cugola, G.; Donadoni, M.; Morassutto, E. “RStream: Simple and Efficient Batch and Stream Processing at Scale”. In: 2021 IEEE International Conference on Big Data (Big Data), 2021, pp. 2764–2774.

[34] Franzén, V.; Östling, C. “Evaluation of Rust for GPGPU high-performance computing”, Master’s Thesis, Chalmers University of Technology and University of Gothenburg, 2022, 77p.

[35] Friedman, E.; Tzoumas, K. “Introduction to Apache Flink: stream processing for real time and beyond”. " O’Reilly Media, Inc.", 2016, 107p.

[36] Gabriel, E.; Fagg, G. E.; Bosilca, G.; Angskun, T.; Dongarra, J. J.; Squyres, J. M.; Sahay, V.; Kambadur, P.; Barrett, B.; Lumsdaine, A.; Castain, R. H.; Daniel, D. J.; Graham, R. L.; Woodall, T. S. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: Proceedings, 11th European PVM/MPI Users’ Group Meeting, 2004, pp. 97–104.

[37] Gherardi, L.; Brugali, D.; Comotti, D. “A java vs. c++ performance evaluation: a 3d modeling benchmark”. In: Proceedings of the Third International Conference on Simulation, Modeling, and Programming for Autonomous Robots, 2012, pp. 161–172.

[38] Gjengset, J. “Rust for Rustaceans: Idiomatic Programming for Experienced Developers”. No Starch Press, 2021, 258p.

[39] Gospodinov, S. “Concurrent Data Processing in Elixir: Fast, Resilient Applications with OTP, GenStage, Flow, and Broadway”. Pragmatic Bookshelf, 2021, 176p.

[40] Griebler, D. “Domain-Specific Language & Support Tool for High-Level Stream Parallelism”, Ph.D. Thesis, Faculdade de Informática - PPGCC - PUCRS, Porto Alegre, Brazil, 2016, 243p.

[41] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. “An Embedded C++ Domain-Specific Language for Stream Parallelism”. In: Parallel Computing: On the Road to Exascale, Proceedings of the International Conference on Parallel Computing, 2015, pp. 317–326.

[42] Griebler, D.; Danelutto, M.; Torquati, M.; Fernandes, L. G. “SPar: A DSL for High-Level and Productive Stream Parallelism”, *Parallel Processing Letters*, vol. 27–01, March 2017, pp. 1740005.

[43] Griebler, D.; Fernandes, L. G. “Towards Distributed Parallel Programming Support for the SPar DSL”. In: Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing, 2017, pp. 563–572.

[44] Griebler, D.; Hoffmann, R. B.; Danelutto, M.; Fernandes, L. G. “Stream Parallelism with Ordered Data Constraints on Multi-Core Systems”, *Journal of Supercomputing*, vol. 75–8, July 2018, pp. 4042–4061.

[45] Griebler, D.; Vogel, A.; De Sensi, D.; Danelutto, M.; Fernandes, L. G. “Simplifying and implementing service level objectives for stream parallelism”, *Journal of Supercomputing*, vol. 76, June 2019, pp. 4603–4628.

[46] Halstead, M. H. “Elements of Software Science (Operating and programming systems series)”. USA: Elsevier Science Inc., 1977, 128p.

[47] He, B.; Zheng, X.; Chen, Y.; Li, W.; Zhou, Y.; Long, X.; Zhang, P.; Lu, X.; Jiang, L.; Liu, Q.; Cai, D.; Zhang, X. “DxPU: Large-scale Disaggregated GPU Pools in the Datacenter”, *ACM Trans. Archit. Code Optim.*, vol. 20–4, Dec. 2023.

[48] Hoffmann, R. B.; Faé, L. G.; Griebler, D.; Li, X. D.; Quintão Pereira, F. M. “Automatic Synthesis of Specialized Hash Functions”. In: Proceedings of the 23rd

ACM/IEEE International Symposium on Code Generation and Optimization, 2025, pp. 317–330.

[49] Hoffmann, R. B.; Faé, L.; Manssour, I.; Griebler, D. “Analyzing C++ Stream Parallelism in Shared-Memory when Porting to Flink and Storm”. In: International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW), 2023, pp. 1–8.

[50] Hoffmann, R. B.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “Stream Parallelism Annotations for Multi-Core Frameworks”. In: XXIV Brazilian Symposium on Programming Languages (SBLP), 2020, pp. 48–55.

[51] Hoffmann, R. B.; Löff, J.; Griebler, D.; Fernandes, L. G. “OpenMP as Runtime for Providing High-Level Stream Parallelism on Multi-Cores”, *Journal of Supercomputing*, vol. 78–6, apr 2022, pp. 7655–7676.

[52] Holk, E.; Pathirage, M.; Chauhan, A.; Lumsdaine, A.; Matsakis, N. D. “GPU programming in rust: Implementing high-level abstractions in a systems-level language”. In: Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum, IPDPSW 2013, 2013, pp. 315 – 324, cited by: 15.

[53] ISO. “Information technology – Programming languages – C”, ISO 9899:2018, International Organization for Standardization, Geneva, Switzerland, 2018, 520p.

[54] ISO. “Programming languages – C++”, ISO 14882:2024, International Organization for Standardization, Geneva, Switzerland, 2024, 2104p.

[55] Jankowski, M.; Pathirana, P.; Allen, S. “Storm Applied: Strategies for real-time event processing”. Simon and Schuster, 2015, 280p.

[56] Jung, R. ““What The Hardware Does” is not What Your Program Does: Uninitialized Memory”. Source: https://www.ralfj.de/blog/2019/07/14/uninit.html, July 2019.

[57] Jung, R.; Dang, H.-H.; Kang, J.; Dreyer, D. “Stacked borrows: an aliasing model for Rust”, *Proc. ACM Program. Lang.*, vol. 4–POPL, Dec. 2019.

[58] Kaiser, H.; Diehl, P.; Lemoine, A. S.; Lelbach, B. A.; Amini, P.; Berge, A.; Biddiscombe, J.; Brandt, S. R.; Gupta, N.; Heller, T.; Huck, K.; Khatami, Z.; Kheirkhahan, A.; Reverdell, A.; Shirzad, S.; Simberg, M.; Wagle, B.; Wei, W.; Zhang, T. “HPX - The C++ Standard Library for Parallelism and Concurrency”, *Journal of Open Source Software*, vol. 5–53, 2020, pp. 2352.

[59] Kanopoulos, N.; Vasanthavada, N.; Baker, R. L. “Design of an image edge detection filter using the Sobel operator”, *IEEE Journal of solid-state circuits*, vol. 23–2, 1988, pp. 358–367.

[60] Khronos Group. “The OpenCL Specification”. Source: https://registry.khronos.org/OpenCL/specs/3.0-unified/html/OpenCL_API.html, January 2025.

[61] Khronos Group. “SPIR-V Specification”. Source: https://registry.khronos.org/SPIR-V/specs/unified1/SPIRV.html, January 2025.

[62] Khronos Group. “Vulkan 1.3.* - A Specification (with all registered extensions)”. Source: https://registry.khronos.org/vulkan/specs/1.3-extensions/html/vkspec.html, January 2025.

[63] Kirk, D. B.; Wen-Mei, W. H. “Programming massively parallel processors: a hands-on approach”. Morgan kaufmann, 2016, 576p.

[64] Klabnik, S.; Nichols, C. “The Rust Programming Language, 2nd Edition”. No Starch Press, 2023, 560p.

[65] Köpcke, B.; Gorlatch, S.; Steuwer, M. “Descend: A Safe GPU Systems Programming Language”, *Proc. ACM Program. Lang.*, vol. 8–PLDI, Jun. 2024.

[66] Laso, R.; Krupitza, D.; Hunold, S. “Exploring Scalability in C++ Parallel STL Implementations”. In: Proceedings of the 53rd International Conference on Parallel Processing, 2024, pp. 284–293.

[67] Lattner, C.; Adve, V. “LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation”. In: Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization, 2004, pp. 75.

[68] Lee, J.; Kim, Y.; Song, Y.; Hur, C.-K.; Das, S.; Majnemer, D.; Regehr, J.; Lopes, N. P. “Taming Undefined Behavior in LLVM”. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2017, pp. 633–647.

[69] Löff, J.; Hoffmann, R. B.; Griebler, D.; Fernandes, L. G. “High-Level Stream and Data Parallelism in C++ for Multi-Cores”. In: XXV Brazilian Symposium on Programming Languages (SBLP), 2021, pp. 41–48.

[70] Löff, J.; Hoffmann, R. B.; Griebler, D.; Fernandes, L. G. “Combining stream with data parallelism abstractions for multi-cores”, *Journal of Computer Languages*, vol. 73, December 2022, pp. 101160.

[71] Löff, J.; Hoffmann, R. B.; Pieper, R.; Griebler, D.; Fernandes, L. G. “DSParLib: A C++ Template Library for Distributed Stream Parallelism”, *International Journal of Parallel Programming*, vol. 50–5, 2022, pp. 454–485.

[72] Manchana, R. “Java Virtual Machine (JVM): Architecture, Goals, and Tuning Options”, *International Journal of Scientific Research and Engineering Trends*, vol. 1, 05 2015, pp. 42–52.

[73] Mattson, T.; Sanders, B.; Massingill, B. “Patterns for Parallel Programming”. Addison-Wesley Professional, 2004, first ed., 355p.

[74] McCool, M.; Reinders, J.; Robison, A. “Structured parallel programming: patterns for efficient computation”. Elsevier, 2012, 432p.

[75] Message Passing Interface Forum. “MPI: A Message-Passing Interface Standard Version 4.1”, 2023, Source: https://www.mpi-forum.org/docs/mpi-4.1/mpi41-report.pdf.

[76] Miri Contributors. “Miri: An interpreter for Rust’s mid-level intermediate representation”. Source: https://github.com/rust-lang/miri, January 2025.

[77] NVIDIA. “CUDA C++ Programming Guide”. NVIDIA, 2024, 540p.

[78] NVIDIA; Vingelmann, P.; Fitzek, F. H. “CUDA, release: 12.6”. Source: https://developer.nvidia.com/cuda-toolkit, January 2025.

[79] Pheatt, C. “Intel® threading building blocks”, *Journal of Computing Sciences in Colleges*, vol. 23–4, 2008, pp. 298–298.

[80] Pieper, R.; Griebler, D.; Fernandes, L. G. “Structured Stream Parallelism for Rust”. In: XXIII Brazilian Symposium on Programming Languages (SBLP), 2019, pp. 54–61.

[81] Pieper, R.; Löff, J.; Hoffmann, R. B.; Griebler, D.; Fernandes, L. G. “High-level and Efficient Structured Stream Parallelism for Rust on Multi-cores”, *Journal of Computer Languages*, vol. 65, July 2021, pp. 101054.

[82] Pieper, R. L. “High-level Programming Abstractions for Distributed Stream Processing”, Master’s thesis, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, 2020, 170p.

[83] Rayon. “Rayon”. Source: https://github.com/rayon-rs/rayon, January 2025.

[84] Rockenbach, D. A. “High-Level Programming Abstractions for Stream Parallelism on GPUs”, Master’s thesis, School of Technology - PPGCC - PUCRS, Porto Alegre, Brazil, 2020, 163p.

[85] Rockenbach, D. A.; Araujo, G.; Griebler, D.; Fernandes, L. G. “GSParLib: A multi-level programming interface unifying OpenCL and CUDA for expressing stream and data parallelism”, *Computer Standards & Interfaces*, vol. 92, March 2025, pp. 103922.

[86] Rockenbach, D. A.; Griebler, D.; Danelutto, M.; Fernandes, L. G. “High-Level Stream Parallelism Abstractions with SPar Targeting GPUs”. In: Parallel Computing is Everywhere, Proceedings of the International Conference on Parallel Computing (ParCo), 2019, pp. 543–552.

[87] Rockenbach, D. A.; Löff, J.; Araujo, G.; Griebler, D.; Fernandes, L. G. “High-Level Stream and Data Parallelism in C++ for GPUs”. In: XXVI Brazilian Symposium on Programming Languages (SBLP), 2022, pp. 41–49.

[88] Rust-GPU. “Rust CUDA Project”. Source: https://github.com/Rust-GPU/Rust-CUDA, January 2025.

[89] Serde Contributors. “Serde: Serialization framework for Rust”. Source: https://github.com/serde-rs/serde, January 2025.

[90] Sydow, S.; Nabelsee, M.; Glesner, S.; Herber, P. “Towards Profile-Guided Optimization for Safe and Efficient Parallel Stream Processing in Rust”. In: 2020 IEEE 32nd International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2020, pp. 289–296.

[91] Tanenbaum, A. S.; Bos, H. “Modern Operating Systems”. USA: Prentice Hall Press, 2014, 4th ed., 1136p.

[92] The Rust Project. “The Rust Reference”. Source: https://doc.rust-lang.org/reference/, January 2025.

[93] The Rust Project. “Rustonomicon: The Dark Arts of Advanced and Unsafe Rust Programming”. Source: https://doc.rust-lang.org/nomicon/, January 2025.

[94] Thies, W.; Karczmarek, M.; Amarasinghe, S. “StreamIt: A Language for Streaming Applications”. In: International Conference on Compiler Construction, 2017, pp. 179–196.

[95] Tokio. “Tokio - The asynchronous runtime for the Rust programming language”. Source: https://tokio.rs, January 2025.

[96] Tronge, J.; Pritchard, H. “Embedding Rust within Open MPI”. In: Proceedings of the SC ’23 Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis, 2023, pp. 438–447.

[97] Tronge, J.; Pritchard, H.; Brown, J. “Improving MPI Safety for Modern Languages”. In: Proceedings of the 30th European MPI Users’ Group Meeting, 2023, pp. 1–11.

[98] Trott, C. R.; Lebrun-Grandié, D.; Arndt, D.; Ciesko, J.; Dang, V.; Ellingwood, N.; Gayatri, R.; Harvey, E.; Hollman, D. S.; Ibanez, D.; Liber, N.; Madsen, J.; Miles, J.;

Poliakoff, D.; Powell, A.; Rajamanickam, S.; Simberg, M.; Sunderland, D.; Turcksin, B.; Wilke, J. “Kokkos 3: Programming Model Extensions for the Exascale Era”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 33–4, 2022, pp. 805–817.

[99] Turon, A.; Vafeiadis, V.; Dreyer, D. “GPS: navigating weak memory with ghosts, protocols, and separation”, *SIGPLAN Not.*, vol. 49–10, Oct. 2014, pp. 691–707.

[100] Veldhuizen, T. “C++ Templates are Turing Complete”. Source: https://www.researchgate.net/publication/2475343_C_Templates_are_Turing_Complete, January 2025.

[101] Virding, R.; Wikström, C.; Williams, M. “Concurrent programming in ERLANG”. Prentice Hall International (UK) Ltd., 1996, 351p.

[102] Vogel, A.; Griebler, D.; Fernandes, L. G. “Providing High-Level Self-Adaptive Abstractions for Stream Parallelism on Multicores”, *Software: Practice and Experience*, vol. 51–6, January 2021, pp. 1194–1217.

[103] Vogel, A.; Rista, C.; Justo, G.; Ewald, E.; Griebler, D.; Mencagli, G.; Fernandes, L. G. “Parallel Stream Processing with MPI for Video Analytics and Data Visualization”. In: High Performance Computing Systems, 2020, pp. 102–116.

[104] Wang, X.; Chen, H.; Cheung, A.; Jia, Z.; Zeldovich, N.; Kaashoek, M. F. “Undefined Behavior: What Happened to My Code?” In: Proceedings of the Asia-Pacific Workshop on Systems, 2012, pp. 1 – 7.

[105] Wang, X.; Zhao, Y.; Pourpanah, F. “Recent advances in deep learning”, *International Journal of Machine Learning and Cybernetics*, vol. 11, 2020, pp. 747–750.

Pontifícia Universidade Católica do Rio Grande do Sul
Pró-Reitoria de Pesquisa e Pós-Graduação
Av. Ipiranga, 6681 – Prédio 1 – Térreo
Porto Alegre – RS – Brasil
Fone: (51) 3320-3513
E-mail: propesq@pucrs.br
Site: www.pucrs.br